BTI1-PTP, Aufgabenblatt 4

Sommersemester 2018

Sichtbarkeit und Lebensdauer, Strings, Rekursion

Ausgabedatum: ..... 23. April 2018

Lernziele

Zeichenketten benutzen können, Rekursive Methodenaufrufe verstehen und einsetzen können.

Kernbegriffe

Jeder Variablen (Exemplarvariable, formaler Parameter, lokale Variable) in einem Java-Programm ist ein *Sichtbarkeitsbereich* (engl.: scope) zugeordnet, in dem sie im Quelltext benutzt werden kann. Er entspricht der Programmeinheit, in der die Variable deklariert ist.

Der Sichtbarkeitsbereich...

- ... einer Exemplarvariablen ist die gesamte Klassendefinition;
- ... eines formalen Parameters ist seine definierende Methode;
- ... einer lokalen Variablen beginnt nach ihrer Deklaration und endet mit dem Ende des Blocks, in dem sie definiert wurde.

Sichtbarkeitsbereiche können ineinander geschachtelt sein, dabei sind Variablen aus umgebenden Bereichen auch in geschachtelten sichtbar; beispielsweise sind die Exemplarvariablen einer Klasse in allen Methoden der Klasse sichtbar, weil Sichtbarkeitsbereiche von Methoden in denen von Klassen geschachtelt sind.

Die *Lebensdauer* (engl.: lifetime) einer Variablen oder eines Objektes ist die Zeit *während der Ausführung eines Programms*, in der der Variablen oder dem Objekt Speicher zugeteilt ist. Lokale Variablen werden beispielsweise auf dem Aufruf-Stack abgelegt und nach Ausführung ihrer Methode automatisch wieder entfernt, sie leben also maximal für die Dauer einer Methodenausführung. Die Lebensdauer einer Referenzvariablen kann sich von der Lebensdauer der referenzierten Objekte unterscheiden.

Zeichenketten bzw. kurz (aus dem Englischen) *Strings* werden in Java als Exemplare der Klasse `java.lang.String` realisiert. Nach der Erzeugung ist ein `String`-Objekt nicht mehr veränderbar. Alle Methoden, die den String scheinbar verändern, liefern in Wirklichkeit ein neues `String`-Objekt zurück und lassen den Original-String unberührt. Daher können Referenzen auf `String`-Objekte bedenkenlos weitergegeben werden. `String`-Objekte können auch durch *String-Literale* (Zeichenfolgen zwischen doppelten Anführungsstrichen) erzeugt werden. Da String-Variablen Referenzen auf `String`-Objekte sind, muss der Vergleich zweier `String`-Referenzen vom Vergleich zweier `String`-Objekte über die Methode `equals` unterschieden werden.

Wiederholungen können alternativ zur Iteration auch mit Hilfe von *Rekursion* (engl.: recursion) definiert werden. Rekursion bedeutet Selbstbezüglichkeit (von lateinisch recurrere = zurücklaufen). Sie tritt immer dann auf, wenn etwas auf sich selbst verweist. In Java kann eine Methode sich selbst aufrufen. Ein rekursiver Aufruf kann direkt erfolgen (direkte Rekursion), oder auch über mehrere Zwischenschritte entstehen (indirekte Rekursion). Um verschiedene (rekursive) Aufrufe derselben Methode besser voneinander unterscheiden zu können, sprechen wir von *Aktivierungen* einer Methode.

Die Entscheidung, ob bei einer bestimmten Problemstellung Rekursion oder Iteration einzusetzen ist, hängt im Wesentlichen davon ab, welche der alternativen Varianten lesbarer und verständlicher ist und welcher Rechen- und Speicheraufwand jeweils mit ihnen verbunden ist.

Der Speicher für die Variablen und Daten eines Java-Programms teilt sich in zwei Bereiche: den *Aufruf-Stack* (engl.: call stack) und den *Heap*. Auf einem allgemeinen Stack werden Elemente nach dem LIFO (Last In First Out) Prinzip verwaltet, d.h. zuletzt abgelegte Elemente werden zuerst wieder entnommen. Der Aufruf-Stack dient zum Ablegen von lokalen Variablen für die Aktivierungen von Methoden: Bei jedem Methodenaufruf werden die Parameter und alle lokalen Variablen auf den Aufruf-Stack geschrieben und erst beim Zurückkehren, z.B. mittels `return`, wieder entfernt. Die maximale Größe des Aufruf-Stacks (*Stack-Limit*) legt fest, wie tief geschachtelt Methodenaufrufe seien dürfen; dies kann bei aufwendigen rekursiven Berechnungen eine Rolle spielen. Wird das Stack-Limit erreicht, bricht die Ausführung mit einer `StackOverflowException` ab.

Der Heap eines Java-Programms dient zum Verwalten der während der Programmausführung erzeugten Objekte. Jedes mittels `new` erzeugte Objekt wird im Heap gespeichert und verbleibt dort mindestens so lange, bis es im Programm keine Referenzen mehr auf das Objekt gibt. Die maximale Heap-Größe begrenzt die Anzahl der Objekte, die in einer Anwendung gleichzeitig existieren können. Wird bei einem bereits vollen Heap ein weiteres Objekt erzeugt, bricht die Ausführung mit einer `OutOfMemoryException` ab.

Aufgabe 4.1 Grundlegende Methoden der Klasse `String`

Im pub-Verzeichnis zu diesem Aufgabenblatt findest du die Archiv-Datei *StringJSE8.zip*. **Entpacke** sie in ein Verzeichnis und öffne die Datei *StringJSE8.htm* in einem Browser. Die angezeigte Dokumentation ist eine abgespeckte Version der Original-Doku der Klasse `String`.

4.1.1 Sieh Dir die Beschreibung der Methoden in der Doku gründlich an. Erzeuge in der Direkteingabe von BlueJ einen beliebigen String, indem Du einfach ein String-Literal eintippst und die Return-Taste betätigst. Links neben der Ausgabe erscheint ein rotes Symbol, das Du in die Objektleiste ziehen kannst; Du erhältst so eine Referenz auf den String. Wie sonst auch in BlueJ, kannst Du nun direkt Methoden an dem Objekt aufrufen. Teste möglichst viele der in der verkürzten Doku aufgeführten Methoden (die anderen kannst Du vorläufig ignorieren), entweder interaktiv in BlueJ oder im Quelltext einer selbstgeschriebenen Klasse.

4.1.2 Such Dir eine der nachgenannten Methoden in der verkürzten Doku aus und erläutere sie bei der Abnahme, idealerweise live in BlueJ mit einem gut vorbereiteten Beispiel.
Zur Auswahl stehen: `charAt`, `compareTo`, `regionMatches`, `indexOf`, `lastIndexOf`, `substring`, `replace`, `trim`.

4.1.3 Benutze die Methoden der Klasse String, um eine Klasse mit dem Namen `DateiNamenZerleger` zu schreiben. Diese Klasse hat die Aufgabe aus einem vollständigen Pfad in Form eines Strings das Verzeichnis, den Dateinamen und die Dateiendung der Datei zu ermitteln. Der ganze Dateipfad und das Trennzeichen sollen beim Erzeugen eines Exemplars übergeben werden. Beispielsweise ist das Zeichen '\' das Trennzeichen in dem Dateipfad „C:\Eigene Daten\Javatest\Beispiel.java". Schreibe drei private Methoden die jeweils die Dateiendung, den Dateinamen und das Verzeichnis ermitteln. Schreibe zusätzlich eine Methode `infoAnzeigen()` für die Ausgabe der Bestandteile.

Lautet zum Beispiel der gesamte Pfad:

```
C:\Eigene Daten\Javatest\Beispiel.java
```

dann soll die Methode `infoAnzeigen()` Folgendes ausgeben:

```
Dateiendung:  java
Dateiname:    Beispiel
Verzeichnis:  C:\Eigene Daten\Javatest
```

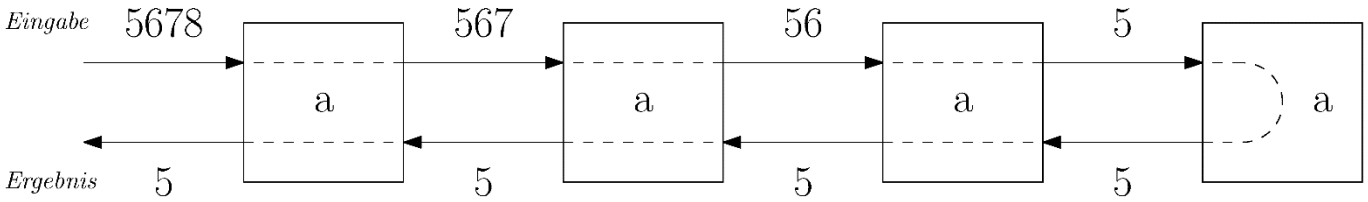
4.1.4 *Zusatzaufgabe*: Beantworte folgende Fragen:

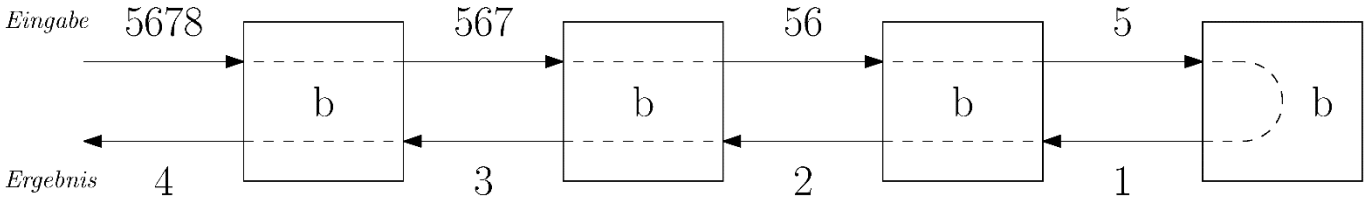
- Wie nützlich sind die beiden aufgeführten Konstruktoren der Klasse `String`?
- Welche Typen tauchen in dieser verkürzten Schnittstelle von `String` auf?
- Was fällt Dir bei den Ergebnistypen aller Methoden auf?

Aufgabe 4.2 Rekursion lesen und verstehen

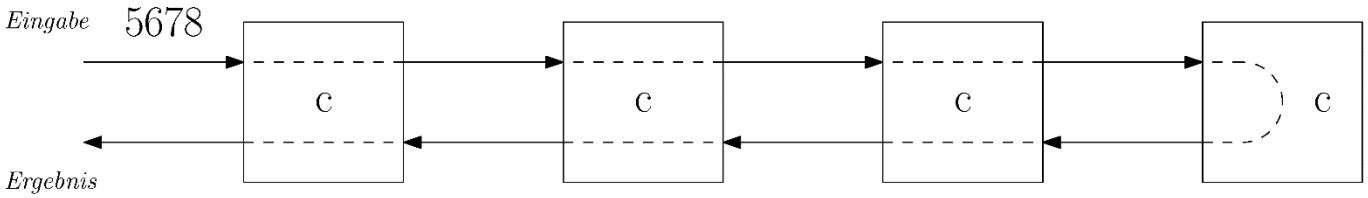
Die Klasse *Konsumieren* des Projekts *Rekursion* beinhaltet einige Methoden, in denen rekursive Algorithmen umgesetzt werden. Die Methoden `a`, `b`, `c`, `d` und `f` sind rekursive Methoden, da sie sich selbst aufrufen. Die zwei Methoden `e` und `g` sind dagegen nicht rekursiv. Sie starten jedoch einen rekursiven Ablauf, indem sie die rekursiven Hilfsmethoden `e2` bzw. `g2` aufrufen.

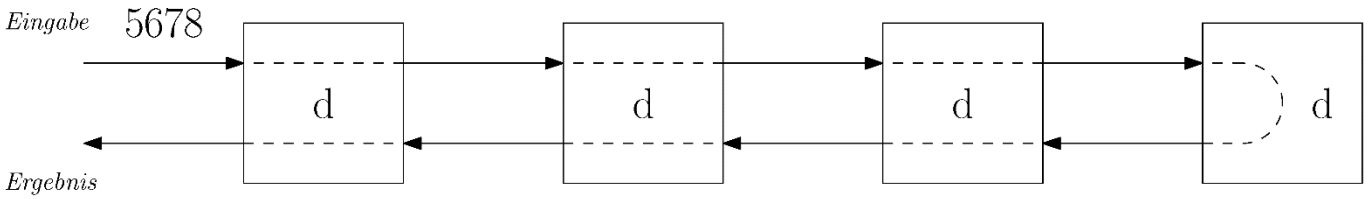
Der Datenfluss durch die Aktivierungen der Methoden kann für konkrete Beispieleingaben anhand von Diagrammen veranschaulicht werden. Für die Aufrufe von `a` und `b` in `testeAlleMethoden` sehen diese wie folgt aus:

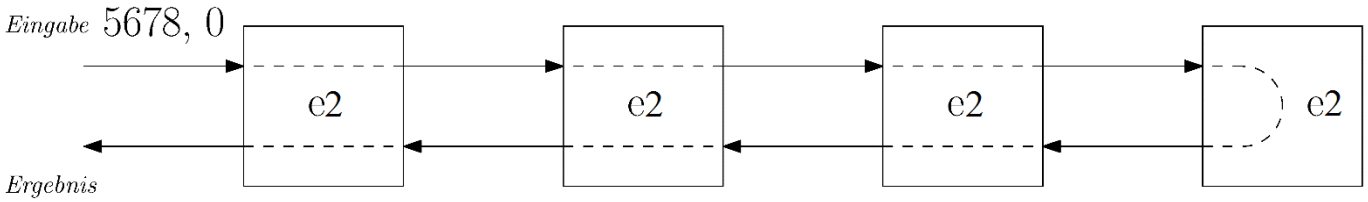


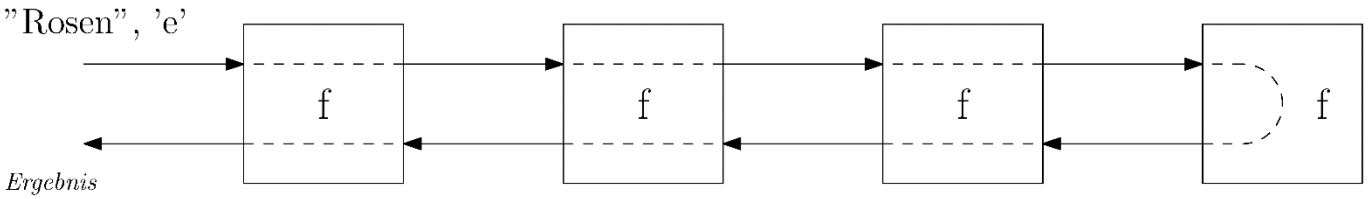


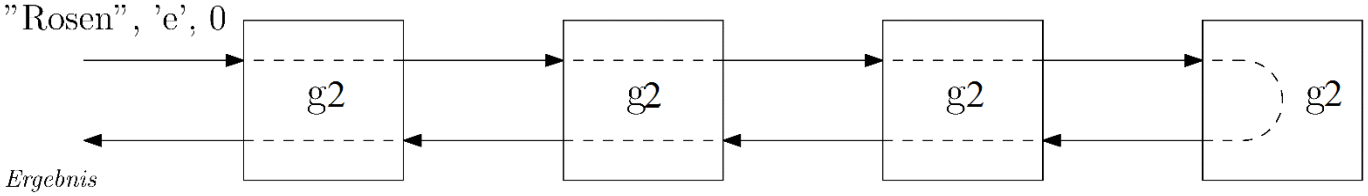
4.2.1 Vervollständige **mindestens vier** der folgenden Diagramme für die verbleibenden fünf Methoden und erkläre bei der Abnahme **mithilfe des Debuggers** den Programmfluss:











4.2.2 Die **Kommentare der Methoden** im Quelltext sind teilweise unvollständig. Ergänze die fehlenden Kommentare im gleichen Stil wie bei den vollständig dokumentierten Methoden: einerseits soll ein **Klient** wissen, was die jeweilige Methode leistet, andererseits soll an einem **Beispiel** die Funktionsweise der Methode deutlich werden.

Aufgabe 4.3 Rekursion aktiv verwenden

4.3.1 Fülle die **Rümpfe** der Methoden (außer der Methode `fak`, die ist zu einfach!) der Klasse `Produzieren` mit *rekursiven* Implementationen. Hier sind keine Schleifen (for, while, do-while) erlaubt! Die kommentierten Beispielauswertungen veranschaulichen mögliche Lösungen.

Für eine Abnahme reichen **vier Methoden** aus, du kannst also eine Methode auslassen.

4.3.2 Schreibe zusätzlich (idealerweise bevor du anfängst, die Rümpfe zu implementieren) eine Methode, die deine Methoden *geeignet* testet (also etwas Besseres als die Methode `testeAlleMethoden` in der Klasse `Konsumieren`). Wie könnte die Methode aussehen, damit sie dir wirklich hilft, Fehler in deinen Methoden zu finden? Wie wählst du die Testfälle aus? Was passiert, wenn ein Test fehlschlägt?

Zusatzaufgabe 4.4 Turtle Graphics

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.

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

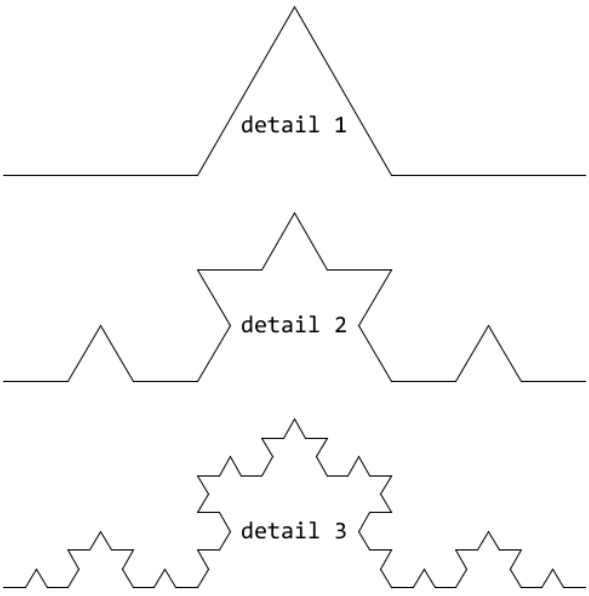
4.4.2 Erweitere nun die Methode so, dass es möglich wird, die Position und Farbe des n-Ecks festzulegen. Da es schnell lästig wird, eine Methode mit mehreren Parametern interaktiv aufzurufen, solltest Du mindestens eine geeignete weitere Methode definieren, die Deine Methode mit passenden Parametern aufruft.

4.4.3 Schreibe eine Methode in `Dompteur`, die kleiner werdende, ineinander geschachtelte n-Ecke zeichnet. Über Parameter soll festgelegt werden können,

- wie viele Ecken die n-Ecke haben sollen,
- wie groß die Kantenlänge des ersten n-Ecks ist.

Es ist nicht notwendig, dass die Rechtecke ineinander zentriert sind!

4.4.4 Implementiere in `Dompteur` die Methode `zeichneKochKurve`, die eine Koch-Kurve mit einem gewissen Detailgrad zeichnet. Die ersten drei Detailgrade sind in der folgenden Grafik veranschaulicht:



Wie man sehen kann, taucht die Detail-1-Kurve in verkleinerter Version 4x in der Detail-2-Kurve auf, und zwar auf 1/3 verkleinert. Die Detail-2-Kurve taucht wiederum in 1/3-Größe 4x in der unteren auf usw.

Allgemein formuliert: um eine Detail-n-Kurve der Länge L zu zeichnen, muss man 4 Detail-(n-1)-Kurven der Länge L / 3 zeichnen. Es bietet sich also an, `zeichneKochKurve` rekursiv zu implementieren:

Wenn der Detailgrad D bei 0 angekommen ist, zeichnet man einfach eine Linie der Länge L. Ansonsten:

- Zeichne Koch-Kurve der Länge L / 3 mit Detailgrad D - 1
- Drehe um 60 Grad nach links (negativ)
- Zeichne Koch-Kurve der Länge L / 3 mit Detailgrad D - 1
- Drehe um 120 Grad nach rechts (positiv)
- Zeichne Koch-Kurve der Länge L / 3 mit Detailgrad D - 1
- Drehe um 60 Grad nach links (negativ)
- Zeichne Koch-Kurve der Länge L / 3 mit Detailgrad D – 1

Danksagung

Die Aufgaben wurden freundlicher Weise von Prof. Dr. Schmolitzky aus früheren Veranstaltungen zur Verfügung gestellt bzw. dem Buch „Java als Erste Programmiersprache“ von Joachim Goll und Cornelia Heinisch entnommen.