PR1, Aufgabenblatt 4

Programmieren I – Wintersemester 2021/22

Strings, Rekursion, Sichtbarkeit und Lebensdauer, Aufzählungen

Ausgabedatum: ..... 22. November 2021

Lernziele

Rekursive Methodenaufrufe verstehen und einsetzen können, Zeichenketten und Aufzählungstypen benutzen können.

Kernbegriffe

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 (per `==` oder `!=`) unterschieden werden vom Vergleich zweier `String`-Objekte über die Methode `equals`.

Neben den formalen Voraussetzungen für die Übersetzbarkeit von Quelltexten gibt es Richtlinien, die nicht durch die Syntax und die Sprachregeln einer Sprache vorgeschrieben werden. Einige sind grundsätzlich geltende *Konventionen*, wie z.B., dass in Java Klassennamen groß und Variablennamen sowie Methodennamen klein geschrieben werden. Diese Konventionen wurden ursprünglich durch die Firma Sun Microsystems (inzwischen von Oracle übernommen) vorgegeben und sollten bei der Softwareentwicklung mit Java eingehalten werden. Andere Konventionen sind projektspezifisch und legen z.B. die Formatierung (Einrückungen etc.) von Java-Quelltext fest. Auch für PR1 gelten solche Quelltext-Konventionen, die eingehalten werden sollten.

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.

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 der Zeitraum *während der Ausführung eines Programms*, in dem 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 kürzer sein als die Lebensdauer des referenzierten Objektes.

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 zur Laufzeit 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.

Ein *Aufzählungstyp* (engl.: enumeration type) ist ein benutzerdefinierter Typ, bei dem die *Wertemenge* bei der Definition des Typs vollständig aufgezählt wird. In Java werden Aufzählungstypen mit dem Schlüsselwort `enum` deklariert. Jeder Enum in Java definiert einige vordefinierte *Operationen*, weitere fachliche Operationen können, wie bei Klassen, über öffentliche Methoden definiert werden.

Aufgabe 4.1 Grundlegende Methoden der Klasse `String` (Termin 1)

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 für Level 2 abgespeckte Version der Original-Doku der Klasse `String`.

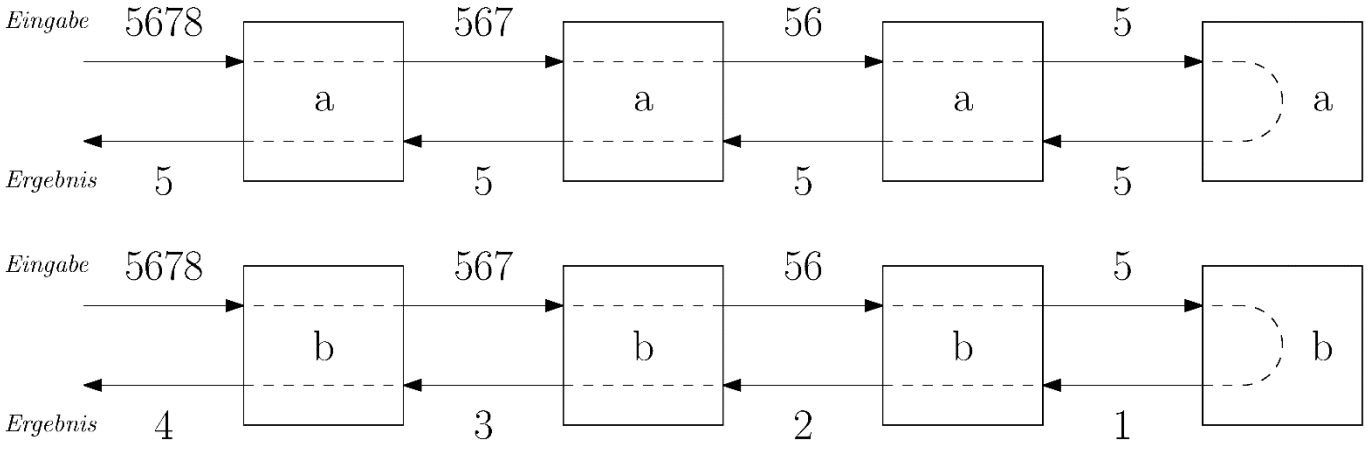
4.1.1 Sieh Dir die Beschreibung der Methoden in der Doku gründlich an. Erzeuge in der Direkteingabe von BlueJ einen beliebigen String, indem Du einfach ein String-Literal eintippst und die Return-Taste betätigst. Links neben der Ausgabe erscheint ein rotes Symbol, das Du in die Objektleiste ziehen kannst; 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**; gern auch mit der Direkteingabe von BlueJ.
Zur Auswahl stehen: `charAt`, `compareTo`, `indexOf`, `lastIndexOf`, `substring`, `replace`, `trim`.

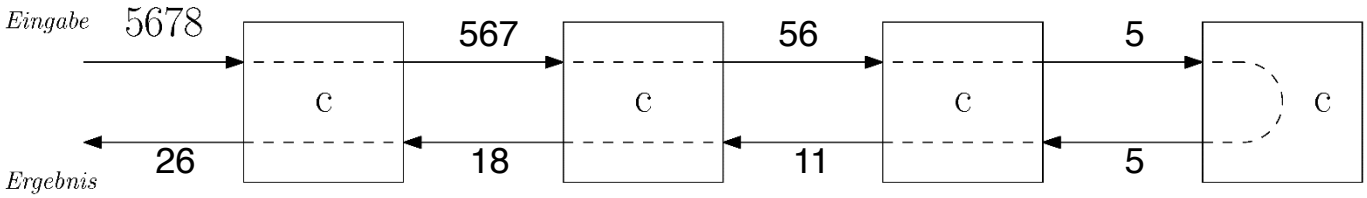
Aufgabe 4.2 Rekursion lesen und verstehen (Termin 1)

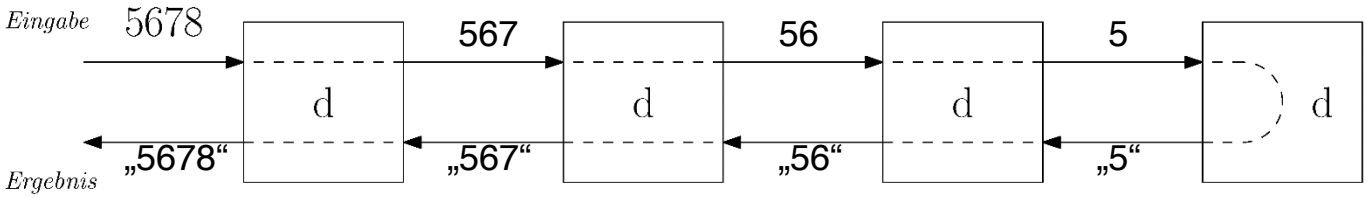
Die Klasse *Konsumieren* des Projekts *Rekursion* beinhaltet einige Methoden, in denen rekursive Algorithmen umgesetzt werden. Die Methoden `a`, `b`, `c`, `d` und `f` sind rekursive Methoden, da sie sich selbst aufrufen. Die zwei Methoden `e` und `g` sind dagegen nicht rekursiv. Sie starten jedoch einen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

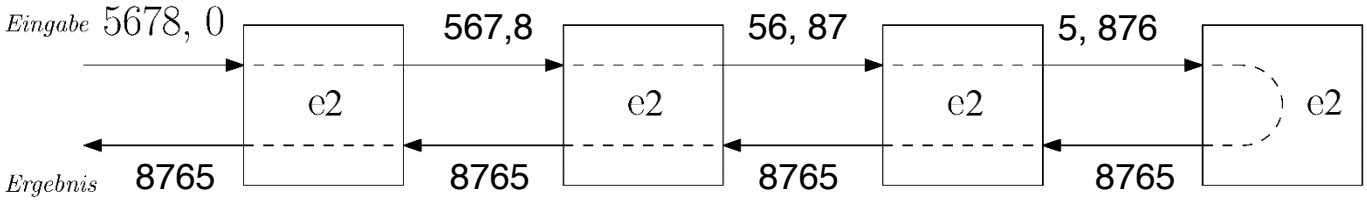
Der Datenfluss durch die Aktivierungen der Methoden kann für konkrete Beispieleingaben anhand von Diagrammen veranschaulicht werden. Für die Aufrufe von `a` und `b` in `testeAlleMethoden` etwa so:

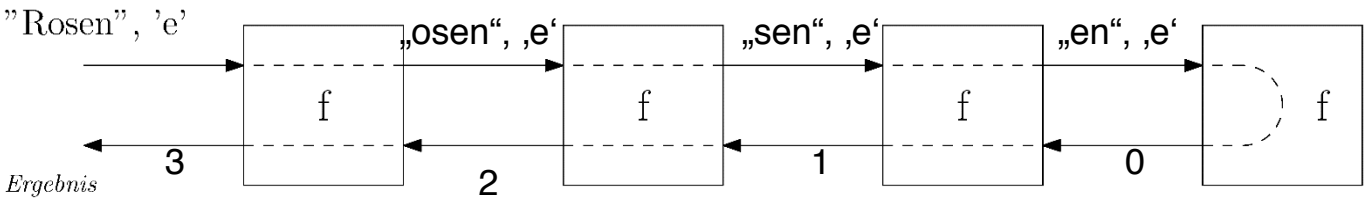


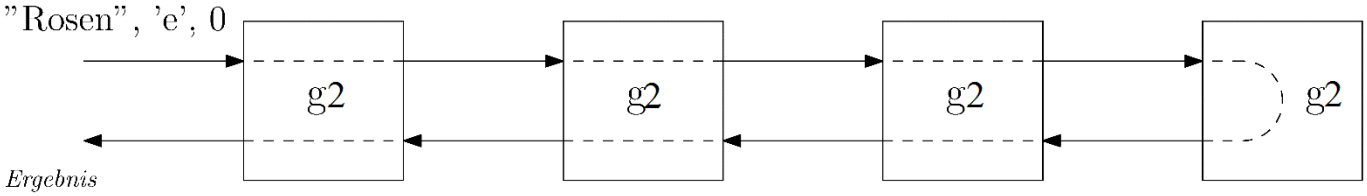
4.2.1 Vervollständige **mindestens vier** der folgenden Diagramme für die verbleibenden fünf Methoden und erkläre bei der Abnahme **mithilfe des Debuggers** den Programmfluss:











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 (Termin 1)

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?

Aufgabe 4.4 Aufzählungen verwenden: Einstieg in die Mau-Mau-Simulation (Termin 2)

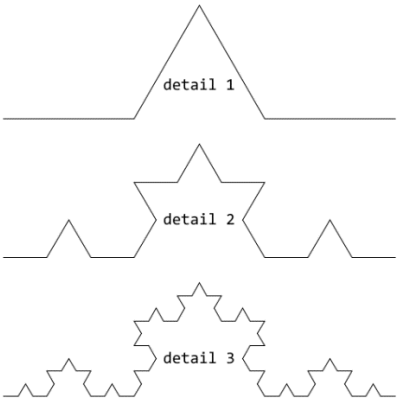
4.4.1 Öffne das Projekt *MauMauSimulation*. Es enthält eine vereinfachte Modellierung einer Runde von Mau-Mau-Spielern. Wenn du Mau-Mau nicht kennst: Lies dir bei Interesse etwas Überblickswissen in der Wikipedia an, hier aber auf jeden Fall den Klassenkommentar in der Dokumentationssicht der Klasse `MauMauRunde` durch. Versuche dann, ihren Quelltext nachzuvollziehen. Ein guter Einstiegspunkt ist ihre Methode `eineSpielrundeSpielen`. Hilfreich können auch die *Dokumentationen* der anderen Klassen sein, von denen du dir testweise Exemplare erstellen kannst. Deren *Quelltexte* (insbesondere der Klassen `Spieler` und `Kartenstapel`) brauchst du dir aber nicht anzusehen.
Welche Einschränkungen hat diese Simulation gegenüber echtem Mau-Mau? **Schriftlich!**

4.4.2 Das Projekt *MauMauSimulation* enthält das geschachtelte Paket `spielkarten` mit einigen Klassen, die in der Simulation benutzt werden. Mit einem Doppelklick auf das Paket-Symbol in BlueJ öffnet sich ein weiteres BlueJ-Fenster mit den Klassen dieses Pakets. Sieh dir die Klassen in der jeweils vorgegebenen Darstellungsform an (einige im Quelltext, andere in ihrer Dokumentation). Was könnte der Grund dafür sein, dass `Spielkarte` und `Kartenbild` getrennt modelliert wurden? **Schriftlich!**

4.4.3 Immer nur Mau-Mau spielen ist langweilig, zwischendurch macht die Runde auch andere Sachen. Implementiere in der Klasse `MauMauRunde` eine neue (von den anderen unabhängige) Methode `zieheDreiBilder`. Diese soll sich einen frischen Kartenstapel erzeugen und von diesem so lange Karten ziehen, bis drei *Bilder* (Bube, Dame oder König) gezogen wurden. Die Methode soll diese drei Karten in einem neu erzeugten Objekt einer Klasse `KartenTripel` zurückliefern, die du geeignet für diesen Zweck selbst definieren musst.

Aufgabe 4.5 Turtle Graphics rekursiv (Termin 2)

4.5.1 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:



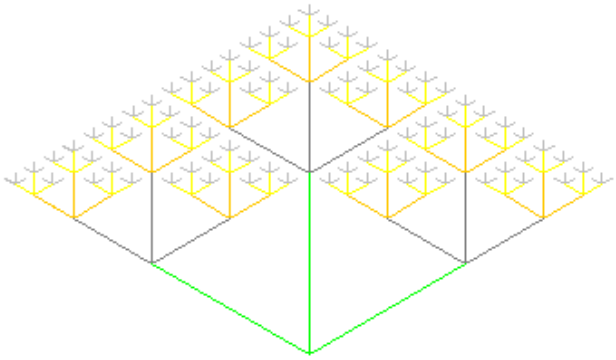
Wie man sehen kann, taucht die Detail-1-Kurve in verkleinerter Version 4x in der Detail-2-Kurve auf, und zwar auf 1/3 verkleinert. Die Detail-2-Kurve taucht wiederum in 1/3-Größe 4x in der unteren auf usw.

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

4.5.2 Schreibe eine Methode in `Dompteur`, die einen Baum zeichnet. Per Parameter soll die Anzahl der Äste, die aus jeder Gabelung entstehen, festgelegt werden. Auch soll die Position, an der der Baum gezeichnet wird, und die Länge der ersten Äste angegeben werden können. Die Äste sollen pro Gabelung kürzer werden, bis ein Minimalwert erreicht ist. Zusätzlich soll die Farbe der Äste variieren. Ein Beispiel für eine mögliche Umsetzung:



Hinweise: Implementiere die Methode rekursiv, so dass für jede Gabelung ein neuer Methodenaufruf erfolgt. Als Abbruchbedingung der Rekursion kann das Unterschreiten einer bestimmten Astlänge dienen. Die Äste können in einem ersten Schritt iterativ gezeichnet werden.