Inhaltsverzeichnis

Variationen zum Thema: Java Eine spielerische Einführung 2

Karel....4
Review 11
Projekte 11
Fragen 14
Referenzen 14

Graphics....15
Review 20
Projekte 21
Fragen 24
Referenzen 24

Console....25
Review 37
Projekte 38
Fragen 41
Referenzen 43

Agrar....45
Review 53
Projekte 54
Challenges 59
Fragen 64
Referenzen 65

MindReader....67
Review 74
Projekte 75
Challenges 82
Fragen 84
Referenzen 85

Swing....87
Review 94
Projekte 95
Challenges 98
Fragen 99
Referenzen 99

Asteroids....101
Review 109
Projekte 109
Challenges 119
Fragen 130
Referenzen 130

Stocks....131
Review 141
Projekte 141
Fragen 158
Referenzen 160
Epilogue 161

....161

Variationen zum Thema:
Java
Eine spielerische Einführung

von Ralph P. Lano, 1. Auflage

Für wen

Für Sie, die lieben Leser. Wenn Sie bisher noch keine oder wenige Programmierkenntnisse haben, dann sind Sie hier richtig. Auch wenn Sie Bücher mit vielen Bildern mögen, sind Sie hier richtig. Allerdings lassen Sie sich nicht durch die Seitenanzahl täuschen: es kommt viel Arbeit auf Sie zu. Wenn das eher nix für Sie ist, dann sind Sie hier nicht richtig.

Von wem

Ich bin seit 2011 Professor für Internetprogrammierung und Multimediaapplikationen im Studiengang MediaEngineering an der Technischen Hochschule Nürnberg. Von 2003 bis 2010 war ich Professor für Softwaretechnik und multimediale Anwendungen an der Hochschule Hof, und von 2010 bis 2011 Professor für Media and Computing an der Hochschule für Technik und Wirtschaft Berlin. Ich promovierte 1996 an der University of Iowa zum Thema 'Quantum Gravity: Variations on a Theme'. Von 1996 bis 1997 war ich Postdoctoral Research Associate am Centre for Theoretical Studies des Indian Institute of Science. In der Zeit von 1997 bis 2003 war ich zunächst bei Pearson Education und später bei der Siemens AG in der Softwareentwicklung und dem Projektmanagement tätig.

Über was

Das wird jetzt ein bisschen technisch, aber manche Leute wollen wissen worauf sie sich einlassen: Im ersten Kapitel unternehmen wir mit Karel die ersten Programmierschritte. Dabei lernen wir Methoden, Schleifen, Bedingungen und unsere ersten Software Engineering Prinzipien kennen. Vor allem das Top-Down Prinzip wird das erste Mal vorgestellt. Das zweite Kapitel führt in die Grafikprogrammierung ein, es werden die Klassen der ACM Bibliothek vorgestellt und wir lernen Objekte zu benutzen. Danach folgt eine Einführung in Konsolenprogramme. Dabei lernen wir Variablen und Operatoren kennen, Bedingungen und Schleifen werden vertieft, boolsche Variablen werden vorgestellt und der 'Loop and a Half' wird gezeigt. Im Kapitel Agrar wird das Top-Down Prinzip vertieft und Methoden werden auf ein neues Fundament gestellt. Auch wird gezeigt wie Animationen und MouseEvents funktionieren, und der RandomGenerator wird das erste Mal benutzt. Kapitel fünf beschäftigt sich mit Strings und dem StringTokenizer, und führt Klassen ein. Das Swing Kapitel führt in die Grundlagen für grafische Benutzeroberflächen ein und konkretisiert den Begriff der Instanzvariablen. Die Grundlagen der Objektorientierung, Vererbung und Komposition werden im folgenden Kapitel ausführlich besprochen und an zahlreichen Beispielen vertieft. Außerdem werden Arrays und einfache Bildverarbeitung eingeführt. Nebenbei werden auch KeyEvents angesprochen. Im letzten Kapitel folgen dann Dateien und Fehlerbehandlung, eine Einführung von ArrayList und HashMap, und wir schließen mit Interfaces und dem Begriff Polymorphismus ab.

Wie

lernt man Klavier spielen? Nicht durchs Zuschauen oder Zuhören. Genauso ist es mit dem Programmieren: man muss viel üben! Und je mehr man übt, desto besser wird das mit dem Programmieren. Wobei wir hier nicht nur Programmieren lernen, sondern eigentlich auch erste Schritte im Software Engineering unternehmen. Aber wie Johann Sebastian Bach so schön gesagt hat: „Alles, was man tun muss, ist, die richtige Taste zum richtigen Zeitpunkt zu treffen. “

Die Veranstaltung so wie ich sie unterrichte besteht aus drei Komponenten: der Vorlesung, der Übung und Hausaufgaben. Die Vorlesung ist vier Stunden pro Woche und entspricht jeweils dem ersten Teil eines Kapitels im Buch. Ein Kapitel schaffen wir in ca. ein bis zwei Wochen. In den Übungen, die auch vier Stunden pro Woche dauern, widmen wir uns dann den Projekten. Dabei schaffen wir zwischen vier und sechs der Projekte pro Übung. In der Übung arbeiten die Studierenden in Teams, meist zu zweit, um sich gegenseitig zu helfen.

Die Hausaufgaben werden im wöchentlichen Rhythmus bearbeitet und benötigen ca. 8 bis 10 Stunden pro Woche. Es ist wichtig, dass die Studierenden alleine an der Hausaufgabe arbeiten. Dabei lege ich ganz großen Wert darauf, dass die Studierenden sich am Anfang der Veranstaltung schriftlich dazu verpflichten den Ehrenkodex der Stanford University einzuhalten. Die Hausaufgabe muss auch nicht vollständig bearbeitet sein, oder perfekt funktionieren. Die Hausaufgaben werden individuell in den Übungen besprochen, und fast alle Probleme lassen sich innerhalb von fünf bis zehn Minuten lösen.

Wo

finde ich die Beispiele und den Quellcode? Die gibt es auf der Webseite zum Buch: www.VariationenZumThema.de. Auch Updates, Links zur Entwicklungsumgebung, das Buch in elektronischer Version gibt's dort. Das Buch selbst gibt's bei Amazon, in Schwarz-Weiß (billig) und in Farbe (teuer).

Darf ich

die Beispiele verwenden, oder das Buch kopieren? Dieses Material steht unter der Creative-Commons-Lizenz Namensnennung - Nicht-kommerziell - Weitergabe unter gleichen Bedingungen 4.0 International (CC-BY-NC-SA 4.0) D.h. Sie dürfen das Material in jedwedem Format oder Medium vervielfältigen und weiterverbreiten, das Material remixen, verändern und darauf aufbauen. Aber Sie müssen angemessene Urheber- und Rechteangaben machen, einen Link zur Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden. Diese Angaben dürfen in jeder angemessenen Art und Weise gemacht werden, allerdings nicht so, dass der Eindruck entsteht, der Lizenzgeber unterstütze gerade Sie oder Ihre Nutzung besonders. **Sie dürfen das Material nicht für kommerzielle Zwecke nutzen.** Und wenn Sie das Material remixen, verändern oder anderweitig direkt darauf aufbauen, dürfen Sie Ihre Beiträge nur unter derselben Lizenz wie das Original verbreiten und Sie dürfen keine zusätzlichen Klauseln oder technische Verfahren einsetzen, die anderen rechtlich irgendetwas untersagen, was die Lizenz erlaubt. Um eine Kopie dieser Lizenz zu sehen, besuchen Sie http://creativecommons.org/licenses/by-nc-sa/4.0/.

Der Quellcode steht unter der MIT License (http://choosealicense.com/licenses/mit/).

Warum

dieses Buch? Ich halte die Vorlesung "Programmieren 1" seit ca 2004 in verschiedenen Formen, Studiengängen und Hochschulen. Dabei war ich jahrelang sehr unglücklich mit der Art und Weise wie Programmieren allgemein unterrichtet wird. Bis ich 2010 auf die Vorlesung "Programming Methodology" der Stanford University gestoßen bin. Hier haben sich ein paar sehr schlaue Köpfe zusammengetan und über 10 Jahre ein didaktisch fein abgestimmtes Konzept entwickelt, wie man Programmierung in einer lustigen, motivierenden aber auch herausfordernden Art und Weise beibringen kann. Meine Studierenden haben immer wieder genörgelt, ob es nicht ein Buch dazu auf Deutsch gibt. Jetzt gibt's eins.

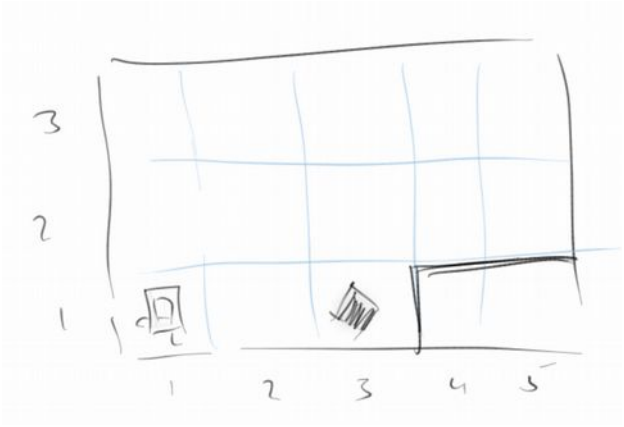
Woher

kommen die Ideen? Ohne Zweifel ist der Stil, der pädagogische Aufbau und viele der Beispiele inspiriert von Mehran Sahami's Vorlesung [1], die ja wiederum auf dem Buch von Eric Roberts [2] basiert. Ohne diese beiden, würde dieses Buch ganz anders aussehen. Auch Robert Sedgewick und Kevin Wayne's Buch [3] hat mich sehr beeinflusst, sowie das Buch von Ägidius Plüss [4]. Ich kann all diese Bücher nur wärmstens empfehlen, alle drei sind diesem hier weit überlegen. Trotzdem bilde ich mir ein, mit diesen Zeilen auch etwas Neues, hoffentlich Nützliches, geschaffen zu haben.

Referenzen

[1] Programming Methodology, CS106A, von Mehran Sahami, https://see.stanford.edu/Course/CS106A
[2] The Art and Science of Java, von Eric Roberts, Addison-Wesley, 2008
[3] Introduction to Programming in Java, von Robert Sedgewick und Kevin Wayne
[4] Ägidius Plüss, Java exemplarisch, http://www.aplu.ch/home/apluhomex.jsp?site=0

Karel



Karel ist ein kleiner Roboter, der etwas wie ein alter Macintosh aussieht. Auf den folgenden Seiten werden wir viel von Karel lernen. Obwohl es etwas einfach anmutet, lernen wir in diesem Kapitel wahrscheinlich die wichtigste Lektion des ganzen Buches, nämlich komplexe Probleme mit dem Top-Down Ansatz in einfachere zu zerlegen.

Ein kleine Anmerkung für Leute die schon ein bisschen programmieren können: man sollte dieses Kapitel auf keinen Fall überspringen! Prinzipiell geht es in diesem Kapitel darum den Top-Down Ansatz an einfachen Beispielen zu erlernen und zu üben, sich einen guten Stil anzugewöhnen, und vor allem zu lernen ohne Variablen zu programmieren.

Karel's Welt

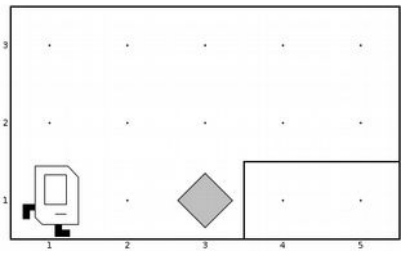
Karel's Welt besteht aus Straßen und Alleen. Straßen verlaufen von West nach Ost und Alleen von Süden nach Norden. Außerdem gibt es Wände durch die Karel nicht hindurchgehen kann, und es gibt Bonbons. Karel hat immer eine Tüte mit unendlich vielen Bonbons bei sich, aber es können auch Bonbons an beliebigen Stellen herumliegen.

Karel kennt von Haus aus vier Kommandos:

- **move():** er kann sich einen Schritt nach vorne bewegen
- **turnLeft():** er kann sich nach links drehen
- **pickBeeper():** er kann einen Bonbon aufheben
- **putBeeper():** er kann einen Bonbon hinlegen

Obwohl das jetzt nicht nach viel aussieht, stellt sich heraus, dass Karel alles ausrechnen kann was es so auszurechnen gibt, man sagt, Karel ist eine 'Universal-Rechenmaschine'.

Karel lernt Laufen



Wenn wir nun möchten, dass Karel den Bonbon aufheben soll und an die Stelle (5,2) tragen soll, so würden wir ihm sagen:

"Karel geh doch zwei Schritte nach vorne, dann hebe den Bonbon auf, drehe dich nach links, laufe einen Schritt, drehe dich dreimal nach links, laufe noch zwei Schritte und lege den Bonbon hin."

Warum soll sich Karel dreimal nach links drehen? Weil er nicht weiß wie 'nach rechts drehen' geht, das hat ihm noch keiner gezeigt.

Da Karel kein Deutsch versteht, müssen wir ihm das in seiner Sprache sagen und da hört sich das wie folgt an:

```
move();
move();
pickBeeper();
turnLeft();
move();
turnLeft();
turnLeft();
turnLeft();
move();
move();
putBeeper();
```

Ähnlich wie niemand wirklich versteht warum man im Deutschen diese komischen Satzzeichen, wie Kommas, Punkte usw. benötigt, weiß auch niemand warum Karel diese runden Klammern und Strichpunkte benötigt. Aber ohne geht's halt nicht. Auch achtet Karel akribisch auf Groß- und Kleinschreibung, wenn man sich vertippt macht er gar nichts.

Karel Programm

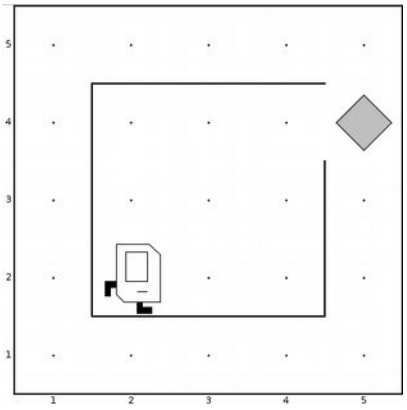
Damit wir nun Karel bei seiner Arbeit zusehen können, benötigen wir etwas das 'Programm' heißt. Dabei handelt es sich eigentlich um Java, aber das brauchen wir noch nicht zu wissen. Ein Karel Programm sieht wie folgt aus:

```
import stanford.karel.*;

public class FirstKarel extends Karel {

    public void run() {
        move();
        pickBeeper();
        move();
```

```
            turnLeft();
            move();
            turnLeft();
            turnLeft();
            turnLeft();
            move();
            move();
            putBeeper();
        }
    }
```

Dabei ist der Teil der für uns wichtig ist blau markiert. In allem was folgt werden wir einfach unsere Karel Kommandos (auch Code genannt) anstelle der blauen Zeilen einfügen.

Übung: GoodMorningKarel



Wir wollen jetzt unser nächstes Karel Problem lösen: Karel ist gerade aufgewacht, und will seine MorgenMilch trinken. Allerdings, steht die noch vor der Haustür. Er muss also aufstehen, hinausgehen, die Milch holen, und sich dann an seinen Frühstückstisch setzen, um in Ruhe seine Milch zu trinken. Wie würde das denn in Karel's Sprache aussehen?

Karel lernt neue Tricks

Karel ist ein bisschen wie ein Haustier, und wir können ihm neue Sachen beibringen. Als erstes wollen wir ihm beibringen, dass er sich auch nach rechts drehen kann. Wir sagen ihm also, jedes mal wenn du das Kommando *turnRight()* siehst, dann drehe dich doch dreimal nach links. Damit uns Karel versteht müssen wir natürlich das ganze in seiner Sprache formulieren:

```
    public void turnRight() {
        turnLeft();
        turnLeft();
        turnLeft();
    }
```

Jetzt können wir einfach dieses neue Kommando verwenden und Karel versteht was er machen soll. Kommandos nennen wir manchmal auch Methoden, z.B. turnRight() ist eine Methode, aber auch move(). Im Allgemeinen werden Methoden immer durch runden Klammern gekennzeichnet.

Frage: Wie müsste denn der Syntax für eine Kommando *turnAround()* lauten, das Karel dazu veranlasst in die entgegengesetzte Richtung zu schauen?

Übung: GoodMorningKarel

Um zu sehen, dass das auch wirklich funktioniert, wollen wir unseren GoodMorningKarel so modifizieren, dass er anstelle dreier turnLeft()'s ein turnRight() verwendet.

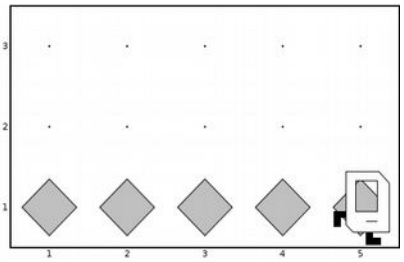
Karel wiederholt sich

Es ist sehr häufig, dass Karel etwas immer wieder machen soll. Z.B. soll er sich dreimal nach links drehen oder er soll zwei Schritte laufen. Für so etwas gibt es etwas das heißt *Schleife*, genauer eine 'for'-Schleife. In Karel's Sprache sieht das dann so aus:

```
    public void turnRight() {
        for (int i=0; i<3; i++) {
            turnLeft();
        }
    }
```

Wichtig ist hier die '3', die sagt Karel, dass er das was in den geschweiften Klammern steht dreimal machen soll. Den Rest brauchen wir vorerst gar nicht zu verstehen. (Die wenigsten Leute tun das, sie kopieren die Zeilen einfach und ändern einfach die '3'.)

Übung: FillRowKarel



Eine andere Anwendung für unsere Schleife ist FillRowKarel. Wir möchten, dass Karel fünf Bonbons in einer Reihe hinlegt. Das scheint eigentlich ein ganz einfaches Problem zu sein, und man würde denken, dass die folgenden Zeilen

```
for (int i = 0; i < 5; i++) {
    putBeeper();
    move();
}
```

das Problem lösen sollten.

Allerdings stellen wir fest, dass Karel am Ende gegen die Wand rennt und sich weh tut. Im Programm sehen wir das wenn das rote Fenster hochpoppt in dem Karel von einer Wespe gestochen wird. Um dies zu vermeiden, könnten wir die Schleife nur viermal durchlaufen. Aber dann würde Karel nur vier Bonbons hinlegen und nicht fünf. Dieses Problem tritt so häufig auf, dass es einen eigenen Namen hat: man nennt es das *O-Bob* Problem.

OBOB

O-Bob war ein Meister der Jedi dem dieses Problem als erstes aufgefallen ist, als er versuchte seinen Droiden zu programmieren. Im Ernst, eigentlich steht OBOB für 'off by one bug', also in etwa 'um eins daneben', meistens eins zu wenig. In unserem FillRowKarel Problem bedeutet dies, dass Karel am Ende einfach noch einen Extra-Bonbon hinlegen muss.

```
for (int i = 0; i < 4; i++) {
    putBeeper();
    move();
}
putBeeper();
```

Momentan müssen wir damit leben. Im nächsten Kapitel werden wir sehen wie man mit Hilfe des 'Loop-and-a-half' das O-Bob Problem elegant umgehen kann.

Karel hat Sensoren

Eine Kleinigkeit haben wir bisher noch verschwiegen, denn Karel hat nämlich Sensoren. Z.B. kann er feststellen, ob an der Stelle an der er sich gerade befindet ein Bonbon liegt. Falls da einer liegen sollte, könnte er diesen aufheben. Dieses 'falls' heißt in Karel's Sprache *if*, und der Syntax der in dazu veranlassen würde einen Bonbon aufzuheben, wenn an der Stelle an der er gerade steht einer liegt, sieht wie folgt aus:

```
if ( beepersPresent() ) {
    pickBeeper();
}
```

Es gibt ne ganze Menge Sensoren, und wir listen hier mal die wichtigsten auf:

- **beepersPresent():** es befindet sich ein Bonbon an der Stelle an der sich Karel gerade befindet
- **noBeepersPresent():** es ist kein Bonbon an der Stelle
- **frontIsClear():** es ist keine Wand vor Karel, Karel kann also unbesorgt vorwärts laufen
- **frontIsBlocked():** es ist eine Wand vor Karel, also sollte Karel nicht versuchen vorwärts zu laufen, sonst haut er sich den Kopf an

Zusätzlich gibt es noch die Sensoren *rightIsClear()* und *leftIsClear()*, sowie die jeweiligen geblockten Varianten. Rechts bedeutet eigentlich unten, und links bedeutet eigentlich oben, immer relativ zur Richtung in der Karel gerade schaut.

Karel, mach mal

Sehr häufig möchten wir, dass Karel *solange* etwas tun soll *bis* ein Ereignis eintritt. Z.B. solange keine Wand vor ihm ist, soll Karel gerade aus gehen. Dafür gibt es das *while* Kommando in Karel's Sprache:

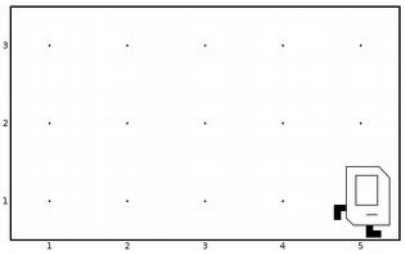
```
while ( frontIsClear() ) {
    move();
}
```

d.h., solange keine Wand vor dir ist, gehe einen Schritt weiter. Man nennt dieses Konstrukt auch die *while* Schleife. Wir sehen an diesem Beispiel auch schön eines unserer ersten Software Engineering Prinzipien (SEP):

SEP: Programme sollen für Menschen lesbar sein und sich wie relativ normales Englisch lesen.

Frage: Wie müsste denn ein neues Kommando *moveToWall()* aussehen, das Karel veranlasst bis zur nächsten Wand zu laufen?

Übung: WallKarel



In WallKarel möchten wir Karel dazu bringen, dass er solange gerade aus weiterläuft, bis eine Wand kommt. Dann soll er aber stehen bleiben, Wir sollten darauf achten unser neues Kommando *moveToWall()* zu verwenden, dann wird der Code nämlich sehr einfach.

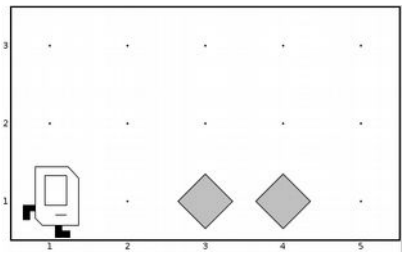
Übung: InfiniteLoopKarel

Die *while* Schleife ist nicht ganz unproblematisch: denn Karel macht etwas solange, bis etwas bestimmtes passiert. Z.B. in InfiniteLoopKarel soll Karel sich solange nach links drehen, bis keine Wand mehr vor ihm ist:

```
while ( frontIsClear() ) {
    turnLeft();
}
```

Das funktioniert gut solange Karel irgendwo in der Nähe einer Wand ist. Befindet sich Karel aber in der Mitte seiner Welt, so hört er nicht auf sich zu drehen. Man nennt das dann eine Endlos-Schleife oder Dreh-Wurm-Karel.

Übung: RobinHoodKarel



Karel liebt Kino. Neulich hat er den Film 'Robin Hood' gesehen. Nach dessen Motto 'nimm von den Reichen und gib es den Armen', läuft Karel so durch seine Welt, und jedes Mal wenn er auf der Straße einen Bonbon findet nimmt er ihn, und wenn kein Bonbon da liegt, legt er einen hin.

Am besten verwenden wir Karel's Sensoren um die Aufgabe zu lösen. Wir können Karel also sagen

```
if ( beepersPresent() ) {
    pickBeeper();
}
if ( noBeepersPresent() ) {
    putBeeper();
}
```

und das Ganze soll er solange machen, bis eine Wand vor ihm ist. Das funktioniert auch ganz gut. Aber wir können den Code etwas verkürzen, wenn wir das *else* Kommando verwenden. Es bedeutet in etwa *andernfalls*. In Code sieht das so aus:

```
if ( beepersPresent() ) {
    pickBeeper();
} else {
    putBeeper();
}
```

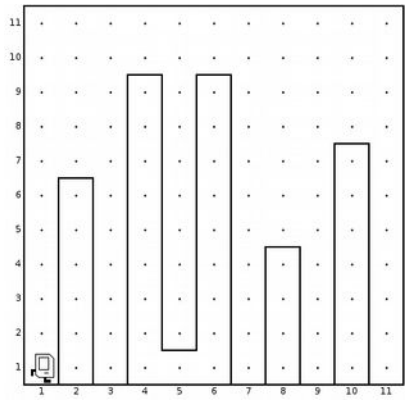
und bedeutet in etwa: wenn ein Bonbon da ist, dann nimm ihn auf, andernfalls lege einen hin.

Top-Down Ansatz

Im Prinzip gibt es zwei Möglichkeiten ein beliebiges Problem zu lösen. Der eine ist der Bottom-Up Ansatz: ausgehend von den Dingen die man kennt versucht man ein Problem zu lösen. Bisher haben wir diesen Ansatz für die Lösung unserer Karel Probleme verwendet: mit den Paar Kommandos die Karel kennt und seinen Sensoren, haben wir einfache Problem einfach Schritt für Schritt gelöst. Für einfache Probleme funktioniert das auch ganz gut.

Für komplexere Problem funktioniert das nicht mehr so gut. Hier hat sich die Top-Down Methode bewährt. In der Top-Down Methode hat man meist ein Problem vor sich, bei dem man zunächst denkt, dass es unmöglich zu lösen ist. Allerdings, gelingt es einem sehr häufig das komplexe Problem in kleinere Teilprobleme zu zerlegen. Manchmal sind diese dann schon lösbar, manchmal muss man aber auch diese noch mal in noch kleinere Teilprojekte zerlegen. Man nennt diesen Prozess 'stepwise refinement'. Betrachten wir das Beispiel des WindowCleaningKarel um zu sehen wie das funktioniert.

Übung: WindowCleaningKarel



Karel lebt in Chicago. Und seinen Lebensunterhalt verdient er sich mit Fensterputzen. Der Job ist nicht ganz ungefährlich, aber mit irgendwas muss er ja sein Geld verdienen. Karel muss pro Woche immer fünf Wolkenkratzer putzen. Er sind aber nicht immer dieselben. Manchmal sind sie höher manchmal niedriger. Eigentlich mag Karel die hohen lieber, sind aber mehr Arbeit.

Wir wollen das Problem mit dem Top-Down Ansatz lösen. Es geht also darum das Gesamtproblem in kleinere zu zerlegen. Nehmen wir an wir wüssten wie Karel *einen* Wolkenkratzer putzt, also angenommen wir hätten eine Methode namens *cleanOneSkyScraper()*. Dann wäre die Lösung unseres Problems ganz einfach, wir würden einfach fünfmal *cleanOneSkyScraper()* ausführen:

```
for (int i = 0; i < 5; i++) {
    cleanOneSkyScraper();
}
```

Wir haben also unser großes Problem gelöst.

Jetzt müssen wir das etwas kleinere Problem *cleanOneSkyScraper()* lösen. Wie putzt denn Karel einen Wolkenkratzer? Noch ist das Problem zu kompliziert um es mit move's und turnLeft's zu lösen, also sollten wir es noch einmal zerlegen. Wenn wir wüssten wie Karel am Wolkenkratzer hoch läuft, *moveUpAndClean()*, über den Wolkenkratzer läuft, *moveOver()*, und an der anderen Seite den Wolkenkratzer wieder hinunterläuft, *moveDownAndClean()*, dann hätten wir das Problem *cleanOneSkyScraper()* gelöst:

```
public void cleanOneSkyScraper() {
    moveUpAndClean();
    moveOver();
    moveDownAndClean();
}
```

Obwohl wir noch nicht ganz fertig sind, sehen wir, dass wir jetzt schon fast am Ziel sind. Die letzten drei Kommandos können wir ganz einfach lösen. Für *moveUpAndClean()* lassen wir Karel sich nach links drehen, dann lassen wir ihn solange laufen, solange die rechte Seite von Karel blockiert ist. Für *moveOver()* lassen wir Karel sich nach rechts drehen, zwei Schritte nach vorne machen und sich dann noch einmal nach rechts drehen. *moveDownAndClean()* ist eigentlich wie *moveToWall()* nur am Ende muss er sich noch einmal nach links drehen.

Empfehlungen für den Top-Down Ansatz

Manchmal ist es z.B. nicht ganz klar wann man mit dem Top-Down Ansatz fertig ist. Hier gibt es ein paar Empfehlungen, die einem da weiterhelfen:

- eine Methode sollte genau ein Problem lösen
- eine Methode sollte nicht mehr als 15 Zeilen lang sein, um die fünf Zeilen sind ideal
- Methoden Namen sollten beschreiben was die Methode macht

Kommentare

Jedes gute Programm hat Kommentare. Je wichtiger es ist, desto mehr Kommentare hat es. Was sind Kommentare? Kommentare sind Beschreibungen was das Programm, was der Code macht. Sie sind für Menschen gedacht, die versuchen das Programm zu verstehen. Der Computer ignoriert die Kommentare.

Der Syntax ist eigentlich ganz einfach: ein Kommentar beginnt mit den Zeichen '/**' und endet mit den Zeichen '*/'. Normalerweise befindet sich ein Kommentar ganz am Anfang eines Programms und erklärt was das Programm als ganzes macht. Zusätzlich sollte man noch bei jeder Methode kurz beschreiben was diese macht. Will man es ganz gut machen, dann beschreibt man auch noch kurz von welchen Annahmen man ausgeht wenn die Methode aufgerufen wird (pre-conditions) und welchem Zustand man die Welt hinterlässt nachdem die Methode fertig ist (post-conditions).

```
/**
 * Karel's day job is to clean windows of skyscrapers in Chicago.
 *
 * @author Ralph P. Lano
 */
public class WindowCleaningKarel extends Karel {

    /**
     * Karel has to clean five skyscrapers, one at a time.
     *
     * PreCondition: Karel is standing in front of the first skyscraper,
     * facing east
     * PostCondition: Karel is behind the last skyscraper, facing east
     */
    public void run() {
        for (int i = 0; i < 5; i++) {
            cleanOneSkyScraper();
        }
    }

    ...

}
```

Hält man die Methoden kurz, so erübrigen sich Kommentare innerhalb von Methoden. Auch hier geht es wieder darum, dass sich Programme wie gutes Englisch lesen sollen.

Übung: Kommentare

Als kleine Übung wollen wir das WindowCleaningKarel Programm ausführlich mit Kommentaren versehen. Für alle zukünftigen Programme die wir schreiben sollten wir uns angewöhnen jedes Programm und jede Methode mit einem kleinen Kommentar zu versehen. Das macht die Programme besser verständlich (für Menschen).

SuperKarel

Bevor wir uns den Projekten widmen, sollten wir noch erwähnen, dass Karel einen großen Bruder namens *SuperKarel* hat. SuperKarel kann alles was Karel kann, aber als großer Bruder kann er natürlich auch turnRight() und turnAround(). Ansonsten sehen die beiden wie Zwillingsbrüder aus. Wenn wir mit Karel's Bruder arbeiten wollen, dann sagen wir einfach 'extends SuperKarel' anstelle von 'extends Karel' am Anfang unserer Programme.

Review

In diesem Kapitel haben wir unsere ersten Programmierschritte unternommen, indem wir Karel bei der Lösung seiner kleinen Problemchen geholfen haben. Außerdem haben wir

- Karel neue Kommandos, auch Methoden genannt, beigebracht, z.B. `moveToWall()`
- Karel mittels der *for* Schleife Sachen wiederholen lassen, z.B. mache drei Schritte vorwärts
- gesehen dass Karel Sensoren hat und dabei die *if* Anweisung kennengelernt, z.B. `if ( beepersPresent() )`
- die *while* Schleife ausprobiert, z.B. `while ( frontIsClear() )`
- Bekanntschaft mit unseren ersten Software Engineering Prinzipien gemacht, z.B. Code sollte sich wie normales Englisch lesen lassen
- unsere ersten Fehler beim Programmieren gemacht, z.B. den OBOB und die Endlosschleife (infinite loop)
- angefangen unseren Code mit `/** ... */` zu kommentieren

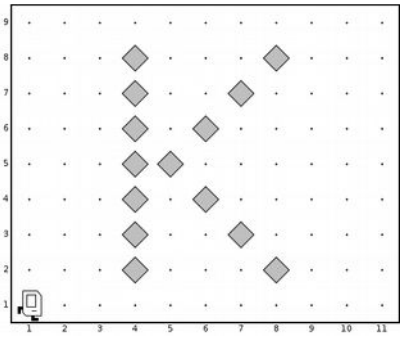
Das wichtigste aber was wir in diesem Kapitel gelernt haben war der Top-Down Ansatz. Dabei geht es darum ein großes Problem in kleinere zu zerlegen, und dies unter Umständen so lange zu wiederholen, bis jedes Teilproblem einfach zu lösen ist. Dieser Ansatz läst sich nicht nur bei Programmierproblemen anwenden, sondern auch bei der Lösung von Problemen in allen möglichen anderen Feldern.

Projekte

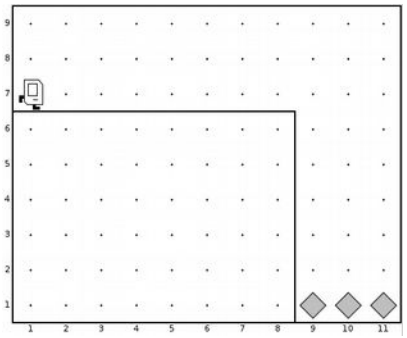
Wir wissen jetzt eigentlich genug über Karel um ihm beim Lösen aller möglicher Probleme zu helfen. In dem Buch von Pavel Solin [4] gibt es noch viel mehr Karel Beispiele.

WritingKarel

Karel hat in der Schule schreiben gelernt. Sein Lieblingsbuchstabe ist das 'K'. Wir wollen also mit Hilfe des Top-Down Ansatzes Karel helfen seinen Lieblingsbuchstaben zu schreiben.

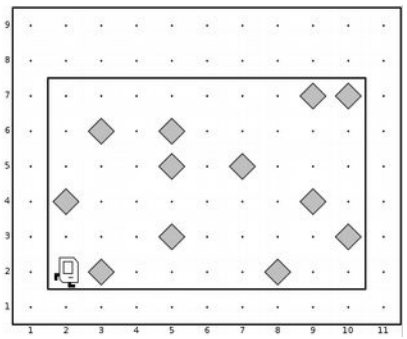


AdventureKarel



Karel ist kein Angsthase, ganz im Gegenteil, er liebt Abenteuer. Aber das kann manchmal ganz gefährlich werden. Neulich war er an den Kreidefelsen von Dover (White Cliffs of Dover). Natürlich will er so nah ran wie möglich. Aber runter fallen will er natürlich auch nicht (Karel kann nämlich nicht schwimmen). Was das Problem schwer macht ist, dass wir nicht wissen wie weit die Klippen entfernt sind.

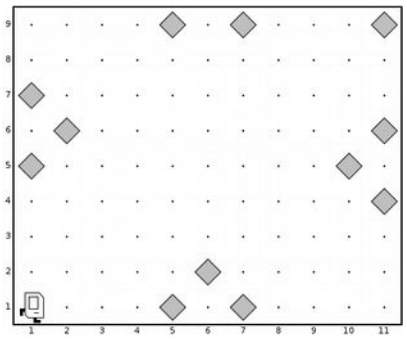
PartyKarel



Karel lebt in einer WG mit seinem Bruder SuperKarel. Beide lieben Parties und nach der letzten sieht es in ihrer Wohnung aus wie Kraut und Rüben. Heute Abend kommen Karel's Eltern zu Besuch, deswegen muss er aufräumen. Helfen wir Karel dabei seine Wohnung wieder in Ordnung zu bringen indem wir alle herumfliegenden Beepers aufräumen.

Der Teil der das Programm etwas kompliziert macht ist dafür zu sorgen, dass Karel auch aufhört wenn er fertig ist.

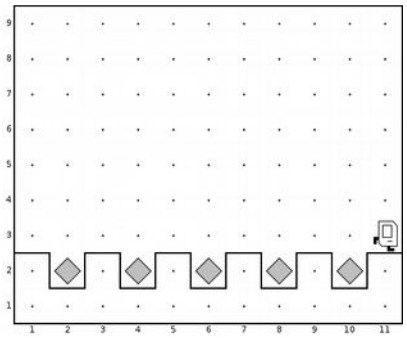
EasterEggKarel



Karel liebt Ostern, da darf er immer Ostereier suchen. Die haben seine Eltern im ganzen Garten versteckt, helfen wir Karel alle Ostereier einzusammeln. Da er später dann auch bei seinen Großeltern vorbeischaut, und deren Garten viel größer ist, muss das Programm auch für Gärten unterschiedlicher Größe funktionieren.

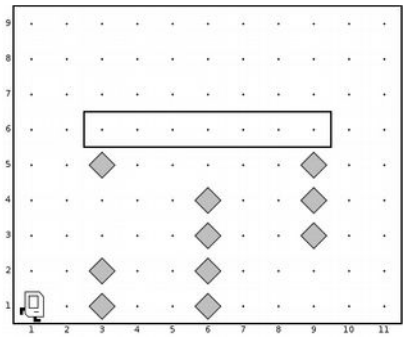
Der Teil der das Programm etwas kompliziert macht ist dafür zu sorgen, dass Karel auch aufhört wenn er fertig ist.

TulipKarel



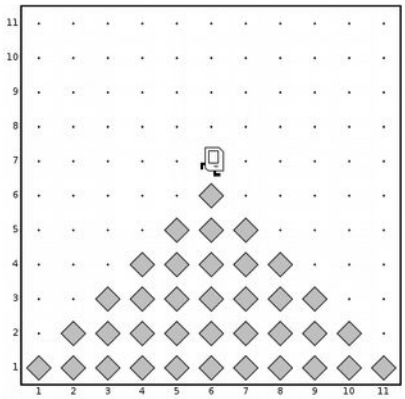
Karel hat holländische Vorfahren, deswegen liebt er Tulpen. Und jeden Winter pflanzt er wieder Tulpenzwiebeln, damit sein Garten im Frühling wieder voller Tulpen ist. Helfen wir Karel beim Tulpenpflanzen, aber mit dem Top-Down Ansatz. Da Karel auch bei seinen Eltern und Großeltern Tulpen pflanzt, muss unser Programm wieder für alle möglichen Gärten funktionieren.

BuilderKarel



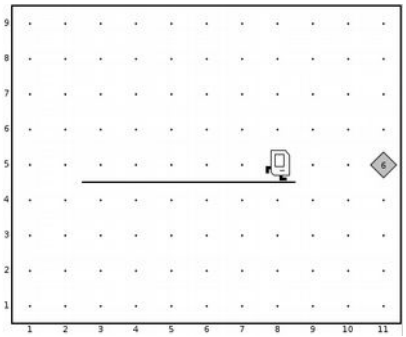
In Chicago gibt's ab und zu Tornados. Der letzte hat Karel's Haus erwischt. Es steht zwar noch aber die Stützpfeiler haben ordentlich was abbekommen und müssen repariert werden. In Karel's Haus gibt es drei Stützpfeiler, die natürlich aus Beepern gemacht sind. Die drei Stützpfeiler sind drei Schritte auseinander, könnten aber unterschiedlich hoch sein. Helfen wir Karel dabei sein Haus wieder zu reparieren, natürlich mit dem Top-Down Ansatz.

PyramidKarel



Wir haben ja schon gehört, dass Karel ein kleiner Abenteurer ist. Neulich hat er im Fernsehen einen Bericht über die Pyramiden gesehen, und da muss er natürlich sofort eine in seinem Garten nachbauen. Da er mehrere bauen will, von unterschiedlicher Größe, sollte unser Programm sollte unser Programm auch in Welten mit anderen Größen funktionieren, deswegen macht es wieder Sinn den Top-Down Ansatz zu verwenden.

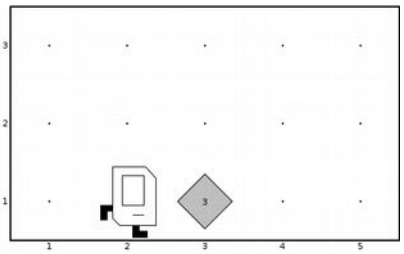
YardstickKarel



Karel ist nicht ganz so dumm wie er ausschaut. Er kann nämlich zählen, obwohl er keine Finger (und Variablen) hat. Karel zählt mit Beepers. In dem Beispiel soll er messen wie lange die Wegstrecke ist vor der er steht. Er soll je nach Länge der Strecke genauso viele Beepers am Ende hinlegen.

Wir können davon ausgehen, dass Karel am Anfang genau vor der Wegstrecke steht.

DoubleBeeperKarel



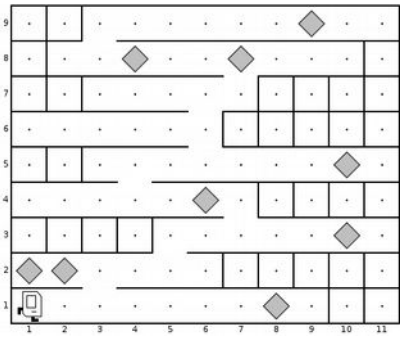
Karel kann auch rechnen. In dem Beispiel DoubleBeeperKarel soll er die Anzahl der Beepers die auf dem Haufen vor ihm liegen verdoppeln. Danach soll Karel wieder vor dem Haufen stehen. Ganz wichtig: Karel kennt keine Variablen, und obwohl er zählen kann (siehe letztes Beispiel) hilft das nicht wirklich. Außerdem soll das Programm für eine beliebige Anzahl von Beepers funktionieren.

Dieses Problem lässt sich am einfachsten mit dem Top-Down Ansatz lösen. Wie wäre es denn wenn man aus einem Beeper zwei machen könnte? Und das macht man solange bis keine Beepers mehr da sind.

Frage: Wie müsste der Code aussehen, wenn Karel die Zahl der Beepers verdreifachen oder halbieren soll?

Karel kann also jede der Grundrechenarten, und damit kann er eigentlich alles ausrechnen was man so ausrechnen kann, man sagt auch Karel ist eine *Universal Computing Machine*.

DonkeyKongKarel



In seiner Freizeit zockt Karel gern. Am liebsten die klassischen Arcadenspiele der 80er. Bei diesen Spielen geht es immer darum so viele Schätze (Beepers) wie möglich einzusammeln und von einem Level zum nächsten zu gelangen. Der Ausgang in diesem Level befindet sich oben rechts und wir dürfen davon ausgehen, dass ein Level aus zehn Stockwerken besteht.

Natürlich verwenden wir wieder den Top-Down Ansatz.

Fragen

1. Nennen Sie die vier wichtigsten Kommandos von Karel.

2. Geben Sie ein Beispiel für ein Off By One Bug (OBOB).

3. In der Übung “PartyKarel” haben Sie Karel das Aufräumen beigebracht. Erklären Sie kurz, wie Karel das Beeper-Chaos wieder bereinigt. (Kein Code nötig, skizzieren Sie lediglich in Worten Ihre Vorgehensweise)

4. Wie 3) nur mit PyramidKarel, DoubleBeepersKarel, WindowCleaningKarel,...

5. In der Vorlesung haben wir Empfehlungen für den Top-Down Ansatz kennengelernt. Diese geben Regeln bzgl. der Namen von Methoden, wie viele Zeilen Code eine Methoden haben sollte, etc. Nennen Sie zwei dieser Richtlinien.

6. Wofür sind Kommentare gut?

7. SuperKarel unterscheidet sich von Karel durch zwei zusätzliche Kommandos. Welche sind das?

8. Was ist der Unterschied zwischen Bottom-Up Design und Top-Down Design. Welcher ist zu bevorzugen?

9. Karel hat den Film 'Robin Hood' angesehen und war total von dem Helden beeindruckt. Deswegen will er seinem Beispiel “nimm von den Reichen und gib es den Armen” folgen. Sie sollen also ein Karel Program schreiben in dem Karel einen Beeper aufnimmt, wenn er einen findet, und einen hinlegt, wenn keiner da ist. Um das Problem zu lösen, dürfen Sie folgende Annahmen über die Welt machen:
   • Die Welt ist mindestens 3x3 groß.
   • Beim Start, steht Karel an der Ecke 1st Street und 1st Avenue, schaut nach Osten (East) und hat unendlich viele Beeper in seiner Bonbontüte.
   Achten Sie darauf, dass Sie nur Karel Kommandos verwenden.

Referenzen

Die Idee hinter Karel stammt von Rich Pattis einem ehemaligen Studenten der Stanford Universität [1]. Der Name 'Karel' ist inspiriert vom Vornamen des tschechischen Schriftstellers Karel Capek in dessen Schauspiel R.U.R. (Rossums Universal-Robots) erstmals das Wort 'Roboter' auftaucht [2]. Mehr Details zu Karel mit vielen Beispielen findet man im Karel Reader [3]. Viele der Beispiele die hier verwendet wurden, stammen zum einen von Karel Reader [3] und zum anderen der Stanford Vorlesung 'Programming Methodologies' [4]. Weitere schöne Karel Beispiele gibt es in dem Buch von Pavel Solin.

[1] Karel the Robot: A Gentle Introduction to the Art of Programming by R.E. Pattis

[2] Seite „Karel Čapek“. In: Wikipedia, Die freie Enzyklopädie. URL: https://de.wikipedia.org/w/index.php?title=Karel_%C4%8Capek&oldid=148374315

[3] KAREL THE ROBOT LEARNS JAVA, von Eric Roberts

[4] CS106A - Programming Methodology - Stanford University, https://see.stanford.edu/Course/CS106A

[5] Learn How to Think with Karel the Robot, von Pavel Solin, http://femhub.com/pavel/work/textbook-1.pdf

Graphics

Grafikprogramme zu schreiben ist fast so einfach wie Karel Programme. In diesem Kapitel werden wir uns mit unseren ersten Grafikprogrammen beschäftigen. Dabei machen wir auch unsere ersten Schritte in Richtung objekt-orientierter Programmierung.

Grafikmodell

Das Grafikmodell das wir verwenden werden ist ganz einfach und erinnert vielleicht an unsere Zeit im Kindergarten, als wir noch mit Filzteilen die tollsten Kunstwerke schufen. Es gibt verschiedene vorgestanzte Formen, man kann aber auch seine eigenen Formen ausschneiden und dann auf einem Filzbrett zu großen Kunstwerken arrangieren.

In unserem 'Filzsortiment' gibt es Rechtecke (GRect), Kreise (GOval), Linien (GLine), Polygone (GPolygon), Bögen (GArc), Bilder (GImage) und Schriften (GLabel). Diese können beliebige Farben und Größen haben und wir können sie an beliebigen Positionen auf unserem Filzbrett anbringen.

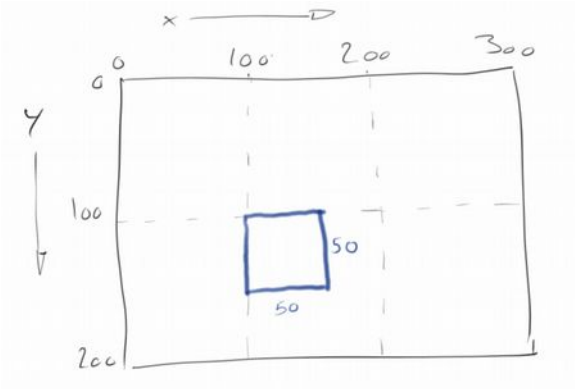
GraphicsProgram

Unser Grundgerüst, also unser Filzbrett sozusagen, ist das *GraphicsProgram*. Das ist vom Prinzip her ganz ähnlich zu einem Karel Programm:

```
import java.awt.Color;
import acm.graphics.*;
import acm.program.GraphicsProgram;

public class House extends GraphicsProgram {

    public void run() {
        // unser Code...
    }
}
```

Die Imports sehen zwar etwas anders aus, und anstelle von 'extends Karel' steht da jetzt 'extends GraphicsProgram'. Uns interessiert das nach wie vor nicht, und wir schreiben unseren Code wie gehabt in die 'run()' Methode.

GRect



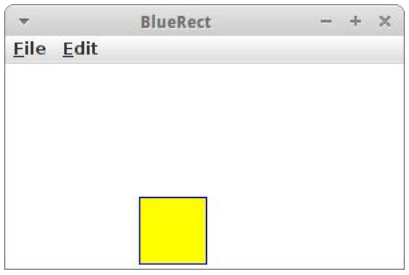
Wir beginnen mit Rechtecken, die wir *GRect* nennen. Ein Rechteck hat eine Breite und Höhe, sowie eine x und y Position. Die folgenden zwei Zeilen zeigen wie das geht:

```
GRect fritz = new GRect(50, 50);
add(fritz, 100, 100);
```

In der ersten Zeile kreiiren wir ein neues (new) Rechteck, das 50 Pixel breit und 50 Pixel hoch ist. Das Rechteck bekommt auch einen Namen, es heißt 'fritz'. Der Name kann fast beliebig sein, und 'fritz' ist so gut wie jeder andere Name. Das ist in etwa so wie wenn wir uns eines der Filzrechtecke herausgesucht hätten und es noch in Händen halten. Wir müssen es aber noch zu unserem Filzbrett hinzufügen (add), und das machen wir mit der zweiten Zeile: wir fügen 'fritz' an der Position (100,100) zu unserem Filzbrett hinzu.

Das Filzbrett heißt eigentlich *Canvas*. Es ist normalerweise etwas mehr als 700 Pixel breit und knapp 500 Pixel hoch. Wir können es aber auch größer oder kleiner machen. Ganz wichtig, x geht wie gewohnt von links nach rechts, allerdings y geht von oben nach unten, das mag vielleicht etwas ungewohnt sein.

Farben



Das Spielen mit Filzteilen wäre ziemlich langweilig wenn alle Filzteile die gleiche Farbe hätten. Genauso ist es mit Grafikprogrammen. Wenn wir also die Farbe unseres Rechtecks ändern wollen, dann setzen (set) wir seine Farbe mit der folgenden Zeile:

```
fritz.setColor(Color.BLUE);
```

In dem Fall wird der Rand von 'fritz' also in blau gezeichnet. Effektiv sagen wir, 'hey fritz, warum setzt du nicht deine Farbe auf blau'. Natürlich versteht 'fritz' nur Englisch (so wie Karel).

Ausgefüllte Rechtecke sind natürlich viel schöner, deswegen können wir 'fritz' noch sagen, dass er doch ausgefüllt sein soll:

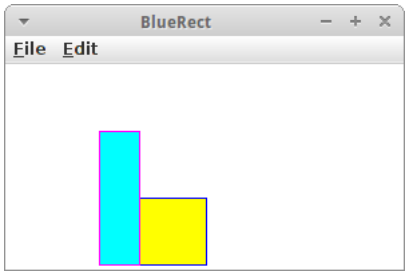
```
fritz.setFilled(true);
```

Manchmal möchte man, dass die Farbe des Randes eine andere ist als die des Inhaltes. Dafür kennt 'fritz' das *setFillColor()* Kommando:

```
fritz.setFillColor(Color.YELLOW);
```

Jetzt ist 'fritz' ein gelbes, ausgefülltes Rechteck mit blauem Rand.

Mehrere Rechtecke



Das Spielen mit Filzteilen wäre auch langweilig wenn es nur ein Filzteilchen gäbe. Wir können beliebig viele andere Rechtecke hinzufügen. Wir müssen nur darauf achten, dass jedes neue Rechteck einen anderen Namen hat:

```
GRect lisa = new GRect(30, 100);
lisa.setColor(Color.MAGENTA);
lisa.setFilled(true);
lisa.setFillColor(Color.CYAN);
add(lisa, 70, 50);
```

Wenn wir diese Zeilen zu unserem Programm oben hinzufügen, dann sehen wir zwei Rechtecke, 'fritz' und 'lisa'.

Nachrichten

Nachrichten, Methoden, Kommandos: verschiedene Namen für das gleiche Konzept. Bei Karel hießen die move() und turnLeft(), jetzt heißen sie setColor() und setFilled(). Wir können unsere eigenen schreiben, wie moveToWall(), oder bereits existierende verwenden. Bei Karel war es etwas einfacher, weil es gab nur einen Karel. Jetzt wird es ein klein wenig komplizierter, denn es kann ja mehr als nur ein Rechteck geben. Deswegen müssen wir immer sagen für wen denn die Nachricht vorgesehen ist, also

```
fritz.setColor(Color.BLUE);
```

ändert die Farbe von 'fritz', aber

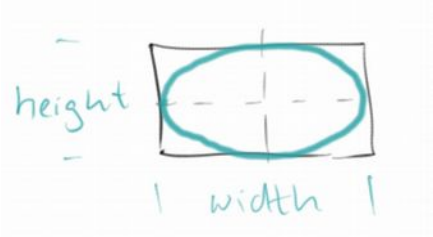
```
lisa.setColor(Color.BLUE);
```

ändert die Farbe von 'lisa'. Also im ersten Fall senden wir die Nachricht an 'fritz', im zweiten an 'lisa'. Welche Nachricht senden wir? Die 'setColor()' Nachricht. Alternativ sagen wir auch, wir rufen die Methode 'setColor()' von 'fritz' auf, oder wir lassen 'fritz' das 'setColor()' Kommando ausführen.

Übung: Flag

Karel wollte schon immer mal verreisen. Um ihn auf seinen großen Trip einzustimmen wollen wir ein GraphicsProgramm schreiben, dass die Flagge des Landes zeichnet in das er gerne reisen würde. (Allerdings mag Karel nur Länder mit einfachen Flaggen!)

GOval



Wenn wir Kreise oder Ellipsen zeichnen wollen dann verwenden wir *GOval*. GOval hat wie ein Rechteck eine Breite und Höhe, und die Ellipse wird sozusagen von diesem Rechteck eingefasst. Ansonsten hat ein GOval die gleichen Methoden wie ein GRect. Auch der Syntax ist identisch:

```
GOval innerRing = new GOval(20, 20);
innerRing.setColor(Color.RED);
innerRing.setFilled(true);
add(innerRing, 125, 125);
```

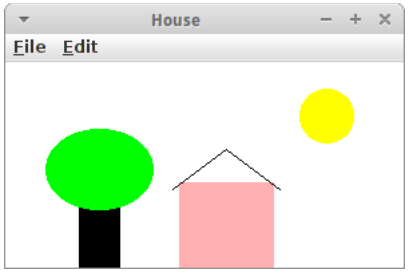
GLine

Um Linien zu zeichnen gibt es die Klasse *GLine*. Man gibt die x und y Koordinaten der beiden Endpunkte, die dann durch eine Linie miteinander verbunden werden.

```
GLine leftRoof = new GLine(100, 150, 125, 125);
add(leftRoof);
```

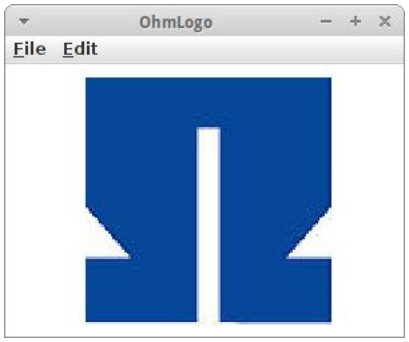
Linien können auch farbig sein. Leider kann man die Dicke von Linien nicht verändern, aber man könnte ja mehrere nebeneinander zeichnen.

Übung: Haus mit Baum



Inzwischen haben wir genügend Grafikkenntnisse um uns an unserem ersten Kunstwerk zu versuchen. Es soll aus einem Baum, einem Haus mit Dach und einer Sonne bestehen. Dafür verwenden wir GOvals, GRects und Glines.

GImage

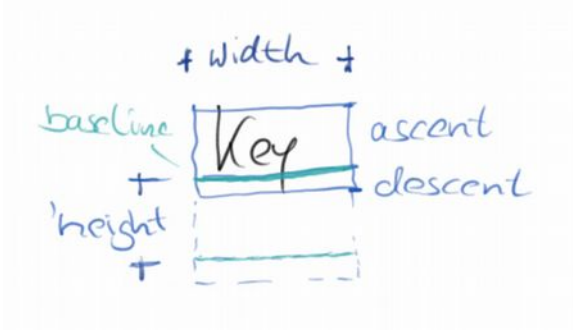


Wirklich praktisch ist die Klasse *GImage*. Mit ihr können wir Bilder zu unserem Canvas hinzufügen:

```
GImage om = new GImage("ohm-logo.gif");
om.scale(2);
add(om, 50, 50);
```

U.a. werden die Formate gif, jpg und png unterstützt, Bilder können auch skaliert werden mit der *scale()* Methode.

GLabel



Ab und zu benötigen wir auch Text in unseren Grafiken, dann kommt der *GLabel* ganz gelegen. Er ist ähnlich einfach zu benutzen wie die anderen Grafikobjekte:

```
GLabel hans =
    new GLabel("Hello World!");
hans.setFont("SansSerif-36");
add(hans, 50, 100);
```

Bei Labels können wir neben der Farbe natürlich auch den Font ändern. Die Zahl gibt die Fontgröße an. Wenn man möchte kann man auch einen kursiven (italic) oder fetten (bold) Font verwenden:

```
hans.setFont("SansSerif-italic-36");
```

Etwas vorsichtig muss man bei der Positionierung sein: die bezieht sich nämlich nicht wie sonst üblich auf die linke obere Ecke, sondern auf die *Baseline*.

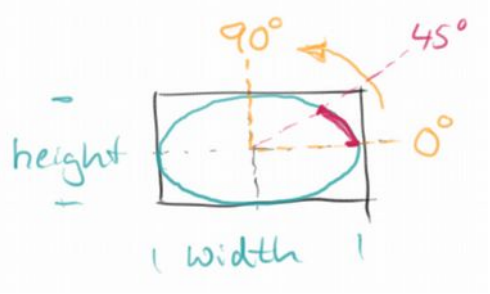
Übung: HelloWorld

Als kleine Übung schreiben wir unser erstes 'Hello World' Programm. Einfach die Zeilen oben in die run() Methode eines GraphicsPrograms einfügen und ausführen.

GArc

GArc zeichnet einen Bogen, also einen Teil eines Kreises oder einer Ellipse. Da aber nur ein Teil des Kreises gezeichnet wird, müssen wir neben der Breite und Höhe der Ellipse auch noch angeben, wo der Bogen beginnen soll und wie lange er gehen soll. Beides wird in Grad angegeben.

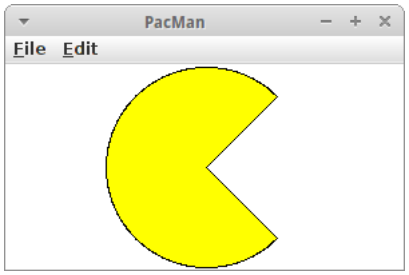
```
GArc archie = new GArc(50, 50, 0, 45);
add(archie, 50, 50);
```



Die obigen Zeilen zeichnen einen Bogen, der bei 0 Grad beginnt und gegen den Uhrzeigersinn 45 Grad umspannt.

In den Projekten werden wir sehen, wie vielseitig Bögen sind und wo sie überall Verwendung finden.

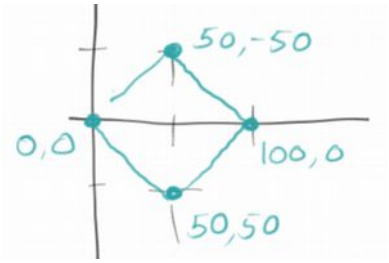
Übung: PacMan



Eine erste praktische Anwendung für einen Bogen ist PacMan. Es ist einfach ein ausgefüllter GArc, der bei 45 Grad beginnt und einen Bogen von 270 Grad umspannt.

GPolygon

GPolygon ist die letzte Grafikklasse die wir noch lernen müssen. Ein Polygon besteht aus Punkten, die man auch Vertices nennt. Mit *addVertex()* fügt man einfach einen Punkt nach dem anderen hinzu. Das Polygon weiß dann selbst wie es die Punkte verbinden muss. Polygone sind immer geschlossen, deswegen kann man sie auch ausfüllen wenn man will.




```
GPolygon diamond = new GPolygon();
diamond.addVertex(0, 0);
diamond.addVertex(50, 50);
diamond.addVertex(100, 0);
diamond.addVertex(50, -50);
add(diamond, 50, 50);
```

Fenstergröße ändern

Manchmal möchten wir die Größe unseres Fensters ändern. Das geht ganz einfach mit der Zeile

```
setSize(300, 200);
```

die am Anfang der run() Methode stehen sollte.

Objektorientierung

Obwohl wir das bisher noch nicht explizit erwähnt haben, haben wir bereits Objekte und Klassen benutzt. Denn fritz, lisa, hans und archie sind Objekte, und GRect, GOval, GLabel usw. sind Klassen. Anfangs ist es nicht immer leicht die beiden zu unterscheiden, aber eigentlich ist es gar nicht so schwer:

- Man sagt z.B. 'fritz ist ein GRect' oder 'archie ist ein GArc'. Man sagt aber nie 'GRect ist ein fritz'.
- Von Objekten kann es mehrere geben, z.B. gibt es einen fritz und eine lisa (beides sind GRects), aber es gibt nur ein GRect.

Um das auch im Code klar zu machen, fangen die Namen von Objekten immer mit Kleinbuchstaben an, während Namen von Klassen immer mit Großbuchstaben beginnen.

SEP: Objekte fangen mit Kleinbuchstaben an, Klassen mit Großbuchstaben.

Review

Was haben wir in diesem Kapitel gelernt? Wir haben

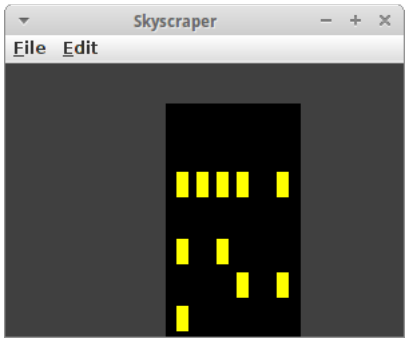
- unser erstes GraphicsProgram erstellt
- die Klassen GRect, GOval, GLine, GImage, GLabel, GArc und GPolygon kennengelernt
- gesehen wie wir Objekten Farbe geben können
- gehört, dass Nachrichten, Methoden und Kommandos dasselbe sind und
- gelernt mehrere Objekte zu benutzen.

Das wichtigste aber in diesem Kapitel war, dass wir erste Schritte in der Objektorientierung unternommen haben.

Projekte

Mit dem was wir in diesem Kapitel gelernt haben, können wir bereits sehr interessante Grafikprojekte realisieren. In dem Buch von Eric Roberts [3] gibt es noch viel mehr Beispiele.

Skyscraper

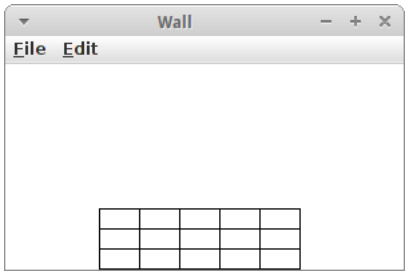


Karel liebt Chicago, und Chicago ist voller Hochhäuser. Wir wollen also für Karel ein Hochhaus bei Nacht zeichnen. Es besteht einfach aus ein paar GRects. Mit der folgenden Zeile

```
setBackground(Color.DARK_GRAY);
```

kann man ganz einfach die Hintergrundfarbe auf dunkelgrau setzen. Wenn man möchte kann man hier eine Methode einführen die *drawWindows()* heißt, und die für das Zeichnen der Fenster verantwortlich ist.

Wall



Als nächstes wollen wir eine Mauer bestehend aus 15 Backsteinen (GRect) errichten. Im Moment ist das noch etwas mühselig, später werden wir sehen, dass es auch einfacher geht.

Archery

Karel geht ab und zu gerne zum Bogenschießen (erinnern wir uns an RobinHoodKarel). Dafür braucht er aber eine Zielscheibe. Diese besteht aus einem inneren roten Ring mit 20 Pixel Durchmesser, einem mittleren weißen Ring mit 40 Pixel Durchmesser und einem äußeren roten Ring mit 60 Pixel Durchmesser. Auch hier kann man wieder mit Methoden arbeiten, damit der Code lesbarer wird (unser erstes SEP), und zwar *drawInnerRing()*, *drawMiddleRing()* und *drawOuterRing()*.

OlympicGames

Bogenschießen ist ja seit 1972 wieder olympische Disziplin, und deswegen hat Karel vor, sich für die nächsten olympischen Spiele zu qualifizieren. Dafür muss er noch viel üben, aber in der Zwischenzeit zeichnen wir schon einmal die olympischen Ringe für ihn.

Es gibt mehrere Ansätze das Problem zu lösen.

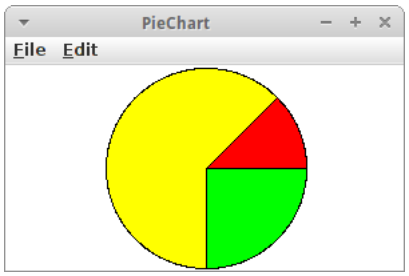
1. Der erste ist nur die Ränder der GOval zu malen, also zu sagen setFilled(false). Dann sind die Ringe aber sehr dünn und schauen nicht ganz richtig aus.

2. Der zweite Ansatz ist je zwei GOvals zu malen, eines etwas größer mit der Farbe und das zweite zentriert darüber mit weißer Farbe. Allerdings werden dann Teile der oberen Ringe komplett überdeckt.
3. Schließlich kann man mehrere Ringe (z.B. fünf) mit Randdicke eins (also wie im ersten Versuch) übereinander malen. Das sieht dann so aus wie in der Grafik hier.

Allerdings ist das immer noch nicht perfekt wenn wir uns das Original ansehen, z.B. in der Wikipedia [7].

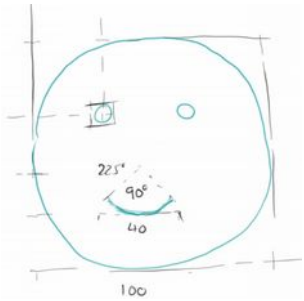
PieChart

Eine schöne Anwendung für GArcs sind Kuchendiagramme (pie-charts). In diesem kleinen Projekt wollen wir ein einfaches Kuchendiagram erstellen, es besteht aus drei GArcs.



Smiley

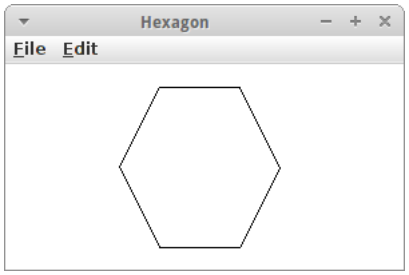
Später benötigen wir ein Smiley. Ein Smiley setzt sich aus einem GOval für das Gesicht, je einem GOval für das linke und das rechte Auge und einem GArc für den Mund. Wenn man kompliziertere Grafiken zeichnet, macht es immer Sinn sich das Design erst einmal auf einem Stück Papier aufzuzeichnen, damit man die Übersicht behält. Man kann dann in der Regel auch ganz einfach die Koordinaten ablesen.



Auch beim Smiley Programm macht es wieder Sinn Methoden zu verwenden, damit der Code besser lesbar ist. Z.B. könnten die Methoden *drawFace()*, *drawLeftEye()*, *drawRightEye()* und *drawMouth()* heißen.

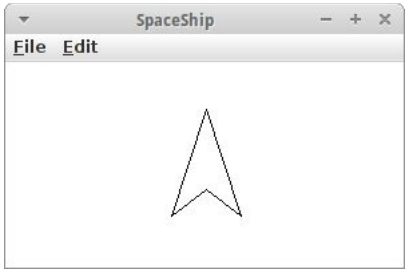
Hexagon

Um uns ein bisschen mit der GPolygon Klasse vertraut zu machen, zeichnen wir ein Hexagon, also ein Sechseck. Sechsecke sind cool, weil die erinnern Karel immer an Bienen, und die machen Karel's Lieblingsessen: Honig.



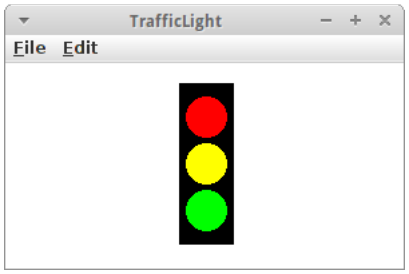
SpaceShip

In Kapitel sieben wollen wir u.a. den Klassiker *Asteroids* programmieren. Dafür benötigen wir ein Raumschiff. Sowohl für das Raumschiff, als auch für die Asteroiden eignet sich das GPolygon. In dieser Übung wollen wir also ein Raumschiff aus einem Polygon konstruieren.



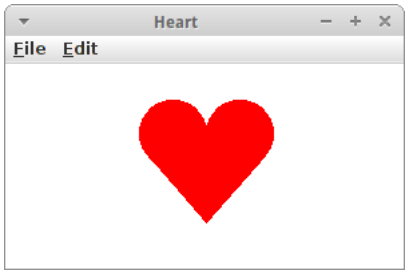
TrafficLight

Eine Ampel besteht aus einem schwarzen Rechteck und aus drei farbigen Kreisen. Das lässt sich ganz einfach aus einem GRect und drei GOvals konstruieren. Später werden wir die Ampel noch animieren.



Heart

Karel möchte gern für seine große Liebe ein Herz zeichnen. Dazu benötigen wir zwei rote GOvals und ein rotes GPolygon. Natürlich sollten die ausgefüllt sein, sonst sieht man ja wie es konstruiert wurde.



CarSymbol

Karel liebt Autos, und natürlich ist Mercedes sein Lieblingsauto. In diesem Projekt wollen wir für unser Lieblingsauto (nicht notwendigerweise Mercedes) ein Logo mithilfe von Grafikobjekten erstellen. Das Mercedes Beispiel besteht aus GOvals und GPolygons.

YinYang

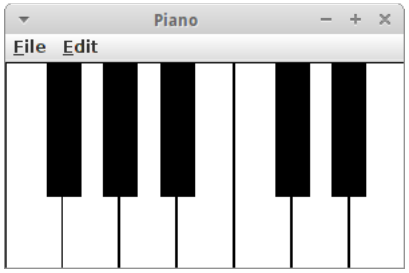
Karel ist fasziniert von Philosophie. Vor ein paar Tagen hat er über Yin und Yang in der Wikipedia gelesen [6]:

"Das höchste Yin ist kalt, das höchste Yang heiß. Kälte entspringt aus dem Himmel, Hitze strömt aus der Erde. Wenn beide einander durchdringen und dabei eine Harmonie erzielen, dann entstehen daraus alle Dinge."

Wir können das YinYang Symbol mithilfe von GOvals und GArcs zeichnen.

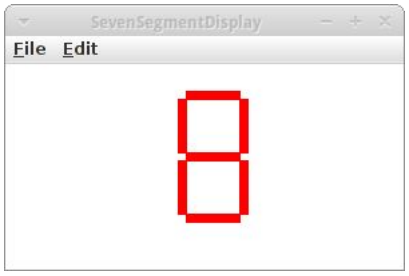
Piano

Natürlich liebt Karel Musik. Und in seiner Freizeit klimpert er immer vor sich hin. Deswegen zeichnen wir eine Klaviatur für ihn, bestehend aus acht weißen und fünf schwarzen Tasten (GRects). Spätestens in Kapitel sieben wird unser Klavier auch Musik machen...



SevenSegmentDisplay

Die Siebensegmentanzeige kommt vor allem in Digitaluhren zum Einsatz um Ziffern anzuzeigen. Die Ziffern werden aus sieben einzelnen "Strichen", auch Segmente genannt, zusammensetzt [8]. Wir können ein Siebensegmentanzeige aus sieben roten Rechtecken erstellen. Später werden wir der Siebensegmentanzeige auch noch Leben einhauchen.



Fragen

1. Mit welcher der Grafik Klassen würden Sie einen Stern zeichnen?
2. In dem folgenden Code, ist 'fritz' ein Objekt oder eine Klasse?
   ```
   GRect fritz = new GRect(50,50);
   ```
3. Was ist der Unterschied zwischen setColor() und setFillColor()?
4. Was sind Nachrichten, wozu dienen diese?
5. Nennen Sie fünf Grafik Klassen die Sie dieses Semester gelernt haben.
6. Was steckt in der acm.jar Datei?
7. Nennen Sie drei Klassen die im 'acm.graphics' Paket zu finden sind.
8. Schreiben Sie Code, der das folgenden Rechteck mit grüner Farbe ausfüllt.
   ```
   GRect fritz = new GRect(50,50);
   ```
9. Was ist ein GPolygon? Geben Sie ein Beispiel für seine Anwendung.
10. Zeichnen Sie das Audi Logo mittels GOvals.

Referenzen

Details zur ACM Grafik Bibliothek kann man auf den Seiten der ACM Java Task Force [1] finden. Dort findet sich auch ein schönes Tutorial, das diese vorstellt [2]. Das Buch von Eric Roberts (der auch hinter der Java Task Force steckt) ist ein Klassiker und eine wahre Schatztruhe, voller Beispiele und tiefen Einsichten. Auch in diesem Kapitel sind viele der Beispiele von seinem Buch und der Stanford Vorlesung 'Programming Methodologies' inspiriert [4].

[1] ACM Java Task Force, cs.stanford.edu/people/eroberts/jtf/

[2] ACM Java Task ForceTutorial , cs.stanford.edu/people/eroberts/jtf/tutorial/index.html

[3] The Art and Science of Java, von Eric Roberts, Addison-Wesley, 2008

[4] CS106A - Programming Methodology - Stanford University, https://see.stanford.edu/Course/CS106A

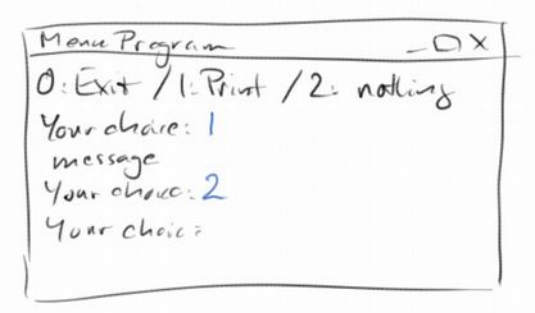
[5] Filz, http://www.desertblossomlearning.com/store/feltstories/images/1047-shapes.jpg

[6] Seite „Yin und Yang“. In: Wikipedia, Die freie Enzyklopädie. URL: https://de.wikipedia.org/w/index.php?title=Yin_und_Yang&oldid=149503099

[7] Seite „Olympic symbols“. In: Wikipedia, Die freie Enzyklopädie. URL: https://en.wikipedia.org/wiki/Olympic_symbols

[8] Segmentanzeige, https://de.wikipedia.org/wiki/Segmentanzeige

Console



Konsolenprogramme sind eigentlich noch einfacher zu schreiben als Grafikprogramme. Aber sie sind etwas abstrakter, deswegen beschäftigen wir uns erst jetzt mit ihnen. Es geht häufig darum mit dem Benutzer unseres Programms zu interagieren. Außerdem werden viele Konzepte die wir in den letzten beiden Kapiteln kennengelernt und verwendet haben, näher erklärt und vertieft.

ConsoleProgram

Schauen wir uns ein einfaches 'HelloWorld' Konsolenprogramm an:

```
import acm.program.ConsoleProgram;

public class HelloWorld extends ConsoleProgram {

    public void run() {
        println("Hello World!");
    }

}
```

Wie üblich interessieren uns nur die Zeilen in der run() Methode. Die besagen, dass der Computer doch den Text in Anführungszeichen, also "Hello World", in einem Konsolenfenster ausgeben soll. Dabei ist 'println' die Kurzform von 'print line' und bedeutet genau das, also schreibe eine Zeile ins Konsolenfenster.

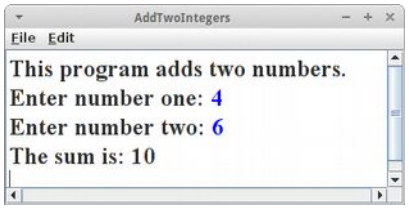
readInt()

Konsolenprogramme wären recht langweilig wenn es nur 'println()' geben würde. Das Gegenstück dazu ist *readInt()*. Es erlaubt dem Benutzer unseres Programms eine Zahl einzugeben.

```
int n1 = readInt("Enter number one: ");
```

Dabei ist der Text in Anführungszeichen nicht unbedingt notwendig, aber er gibt dem Benutzer einen Hinweis darauf was er denn tun soll.

Übung: AddTwoIntegers



Schauen wir uns gleich mal ein Beispiel an. Wir wollen zwei Zahlen addieren und das Resultat auf der Konsole ausgeben.

```
println("This program adds two numbers.");
int n1 = readInt("Enter number one: ");
int n2 = readInt("Enter number two: ");
int sum = n1 + n2;

println( "The sum is: " + sum );
```

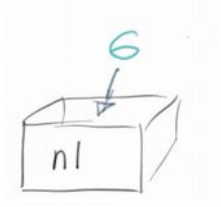
Die erste Zeile teilt dem Benutzer einfach mit was das Programm macht. Dann wird der Benutzer aufgefordert die erste Zahl einzugeben. Das Programm wartet jetzt solange bis der Benutzer eine Zahl eingibt. Danach fordert es den Benutzer auf die zweite Zahl einzugeben. Nachdem der Benutzer dies getan hat, addieren wir die beiden Zahlen *n1* und *n2*. Wir speichern das Resultat in der Variablen *sum*. In der letzten Zeile wird dann die Summe ausgegeben.

Frage: Wie würde wohl ein Programm aussehen, das zwei Zahlen subtrahiert?

Variablen

Die Frage die wir uns sofort stellen, was ist eine *Variable*? Variablen sind ein bischen wie Schachteln in die man Sachen reintun kann. Was für Sachen kann man da reintun? Z.B. Zahlen oder GRects. Das muss man aber immer explizit sagen was in die Schachtel rein kann. Also z.B. in eine Schachtel für Zahlen kann man nur Zahlen rein tun, und in eine Schachtel für GRects kann man nur GRects rein tun. Variablen haben auch immer einen Namen, z.B. 'n1' oder 'fritz'.

Außen auf der Schachtel steht ein Name, z.B. 'n1' oder 'fritz'. Wenn wir also z.B. sagen




```
int n1;
```

dann heißt das es gibt eine Schachtel die den Namen 'n1' hat. Wenn wir dann sagen

```
n1 = 6;
```

dann ist das so wie wenn wir die Zahl 6 in die Schachtel hineintun. Man nennt das auch Zuweisung, also der Schachtel 'n1' wird die Zahl '6' zugewiesen. Wenn man will kann man die Zahl die in der Schachtel ist auch ändern. Man macht dann einfach eine neue Zuweisung und sagt

```
n1 = 5;
```

d.h., wir ersetzen die alte Zahl durch die neue Zahl '5'. Außen auf der Schachtel steht aber nach wie vor 'n1'.

Wir können aber nicht nur Zahlen in die Schachteln tun, sondern auch andere Sachen. Z.B. mit


```
GRect fritz;
```

sagen wir, dass es eine Schachtel gibt, die 'fritz' heißt. In diese Schachtel können wir GRects rein tun, also mit

```
fritz = new GRect(50, 50);
```

legen wir ein neues GRect das 50 Pixel breit und 50 Pixel hoch ist in die Schachtel. Ganz wichtig, in eine Schachtel für GRects können wir keine Zahlen tun, und in eine Schachtel für Zahlen können wir keine GRects tun.

Deklaration und Zuweisung

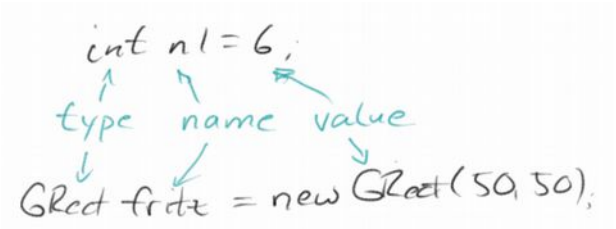
Wenn wir sagen

```
int n1;
```

dann *deklarieren* wir eine Variable. Die Variable heißt 'n1' und ist vom Typ Zahl (int ist eine Abkürzung für Englisch 'integer', was soviel wie Ganzzahl bedeutet). Wenn wir dann sagen

```
n1 = 6;
```

machen wir eine *Zuweisung*, wir weisen also der Variablen 'n1' den Wert '6' zu. Das '=' bedeutet also nicht Gleichheit, sondern Zuweisung. Das ist ganz wichtig.



Was die Namen von Variablen angeht, so können diese fast beliebig sein, sie dürfen aus Buchstaben, Zahlen und dem Unterstrich bestehen. Einige Wörter sind allerdings nicht erlaubt, wie z.B. 'if' und 'for', weil diese bereits von Java verwendet werden. Auch dürfen Namen nicht mit Zahlen beginnen und Umlauten sollte man generell vermeiden. Und man muss immer auf Groß- / Kleinschreibung achten!

Eine Variable hat also immer einen Namen, einen Typ und einen Wert.

SEP: Die Namen von Variablen sollten immer beschreibend sein, also z.B. 'blueRect'.

Typen

Was für Typen gibt es denn? Im letzten Kapitel haben wir schon einige kennengelernt: GRect, GOval, GLine u.s.w. sind alles Typen, genauer gesagt *Datentypen*. Es gibt aber auch andere Typen. Die die uns in diesem Kapitel beschäftigen werden sind die sogenannten *primitiven* Datentypen. Von denen gibt es in Java acht, aber uns interessieren nur die folgenden vier:

- **int:** Ganzzahlen, also 1,2,3, usw. aber auch die 0 und die -5, also positive und negative ganze Zahlen.
- **double:** Gleitkommazahlen, also Zahlen wie 3.14 oder 0.333 usw., (Beim Programmieren verwenden wir wie im Englischen den '.' anstelle des ',').
- **boolean:** wird für logische Werte verwendet, kann nur die zwei Werte *true* oder *false* haben.
- **char:** Buchstaben, wie z.B. 'a', 'b', aber auch Sonderzeichen wie '.' und '$' usw.

Manchmal stellt sich die Frage, soll ich jetzt einen 'int' oder einen 'double' verwenden? Die Antwort ist ganz einfach: kann man etwas zählen, dann verwendet man einen 'int'. Für alles was man nicht zählen kann verwendet man den 'double' Datentyp. Mit dem *boolean* Datentyp beschäftigen wir uns weiter unten, der *char* Datentyp muss noch ein Kapitel warten.

SEP: Für alles was man zählen kann, verwenden wir den Datentyp *int*.

Ausdruck

Das Wort *Ausdruck* wie wir es verwenden, hat nichts mit 'ausdrucken' zu tun. Sondern wird eher im Sinne von 'mathematischer Ausdruck' verwendet. Wir haben schon ein Beispiel gesehen, nämlich

```
int sum = n1 + n2;
```

Hier steht auf der linken Seite der Zuweisung eine Variable, 'sum', und auf der rechten Seite steht ein mathematischer Ausdruck, und zwar die Summe der Zahlen 'n1' und 'n2'. Genauer, die Summe der Zahlen die in den Schachteln 'n1' und 'n2' drinnen sind.

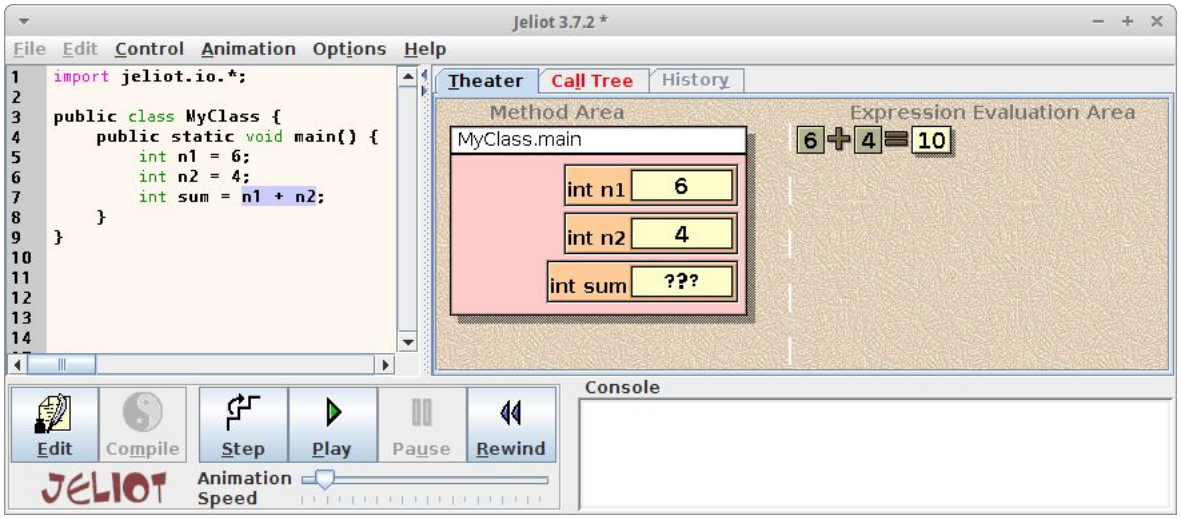
Was dies Zeile also bedeutet ist folgendes: Hole die Zahlen die in der Schachtel 'n1' und in der Schachtel 'n2' stehen, addiere diese und speichere das Resultat in die Schachtel 'sum'. Man kann das sehr schön mit dem Programm 'Jeliot' veranschaulichen [1].

Übung: Jeliot

Jeliot ist ein sehr schönes Programm, dass uns hilft zu visualisieren was passiert, wenn wir die drei Zeilen

```
int n1 = 6;
int n2 = 4;

int sum = n1 + n2;
```

ausführen. Wir sehen links im Codefenster unsern Code. Wir können diesen Code Schritt für Schritt durchlaufen. Und auf der rechten Seite sehen wir einmal den Bereich für die Methoden (Method Area) in dem wir auch unsere Schachteln 'n1', 'n2' und 'sum' sehen. Sowie den Bereich für die Auswertung von Ausdrücken (Expression Evaluation Area). Wir sehen, wie gerade der Ausdruck '6+4=' ausgewertet wird. Im nächsten Schritt wird dann das Resultat ('10') in die Schachtel 'sum' gesteckt.



Operatoren

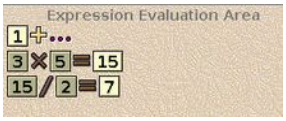
Karel hat Angst vor Operatoren. Er denkt dann immer an Krankenhaus. Wir verwenden das Wort *Operator* wieder eher im mathematischen Sinne, also 'plus', 'minus', 'mal' und 'geteilt durch'. In Java verwenden wir dafür die Zeichen '+', '-', '*' und '/'.

Division

Bei 'plus', 'minus' und 'mal' kann nicht viel schief gehen, die funktionieren immer. Aber bei der Division kann einiges schief gehen. Das erste an das wir uns vielleicht noch erinnern ist Division durch Null: das gibt immer Ärger. Das zweite was Ärger macht ist Ganzzahldivision, also was ist 5 / 2? Damals in der zweiten Klasse war das einfach 2. Aber dann später hat irgendjemand gesagt es ist 2.5. Also was nun?

Das kommt drauf an. Wenn wir mit Ganzzahlen arbeiten, also mit *int*, dann ist es 2. Wenn wir mit Gleitkommazahlen arbeiten, also mit *double*, dann ist es 2.5. Das kann zu interessanten Problemen führen. Nehmen wir das folgende Beispiel:

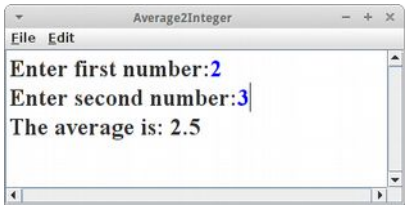
```
int x = 1 + 3 * 5 / 2;
```



Was ist der Wert von 'x'? Sieben oder acht? Mit Jeliot kann man Schritt für Schritt nachvollziehen, wie der Computer das ausrechnet. Zunächst einmal gilt 'Punkt vor Strich', d.h. die Addition wird als letztes ausgeführt. Dann stellt sich aber die Frage wird zu erst '3*5' oder '5/2' ausgerechnet? Hier gilt die Regel von links nach rechts, also wie beim Lesen, man liest (bei uns wenigstens) von links nach rechts. Also wird zuerst '3*5' berechnet, das Ergebnis durch 2 geteilt. Weil es sich um Ganzzahldivision handelt kommt dabei 7 und nicht 7.2 heraus. Wenn wir dann die 7 zu 1 addieren, erhalten wir 8 als Ergebnis.

Was passiert nun wenn wir anfangen Ganzzahlen (int) und Gleitkommazahlen (double) zu mischen? Also, so was wie 5 / 2.0? Dann tut der Computer so wie wenn alle Zahlen Gleitkommazahlen wären.

Übung: AverageTwoIntegers



Man könnte denken, dass Probleme mit der Ganzzahldivision eher selten vorkommen, dem ist aber überhaupt nicht so, und das Beispiel *AverageTwoIntegers* soll das verdeutlichen.

Ähnlich wie bei AddTwoIntegers lesen wir zunächst zwei Ganzzahlen ein. Und wenn jemand etwas nachlässig ist, schreibt er evtl. folgende Anweisung um den Durchschnitt zweier Zahlen zu berechnen:

```
int average = n1 + n2 / 2;
```

Interessanterweise funktioniert das sogar, z.B. wenn n1=1 und n2=3 ist. Allerdings für andere Zahlenwerte merkt man sehr schnell, dass da etwas nicht stimmt.

Natürlich war der Fehler die fehlenden Klammern! Also, versuchen wir es doch mit folgendem Ausdruck:

```
int average = ( n1 + n2 ) / 2;
```

Der funktioniert schon viel besser, aber immer nur wenn n1+n2 eine gerade Zahl ist. Na ja klar, der Fehler liegt im Datentyp: "average" muss ein *double* sein:

```
double average = ( n1 + n2 ) / 2;
```

Aber auch das hilft nicht wirklich. Der Teufel liegt im Detail: da n1 und n2 vom Datentyp *int* sind UND auch 2 vom Datentyp *int* ist, führt der Computer Ganzzahldivision aus. Die Lösung für das Problem lautet:

```
double average = ( n1 + n2 ) / 2.0;
```

Obwohl dieser Fehler im nach hinein offensichtlich ist und die Lösung trivial, tritt er überraschenderweise sehr, sehr häufig auf.

SEP: Man sollte seinen Code immer ausführlich testen.

Punkt vor Strich

Vorrangsegeln wie 'Punkt vor Strich' gibt es ganz viele in Java. Grob gilt folgende Hierarchie:

1. Klammern: ()
2. *, /, %
3. +, -

also zuerst Klammern, dann Multiplikation, Division und Restwert, und schließlich Addition und Subtraktion. Aber es kann noch viel komplizierter werden mit so Operatoren wie '^', '&', '|', usw. Deswegen empfiehlt es sich immer so viele Klammern wie möglich zu setzen.

SEP: Um Ärger zu vermeiden, verwende viele Klammern!

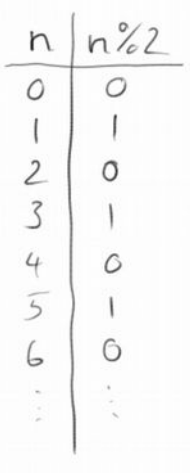
Restwert

Allerdings gibt es in Java noch einen weiteren Operator, den *Restwert* (Englisch: modulo oder genauer remainder). Das letzte mal haben wir wahrscheinlich in der zweiten Klasse davon gehört, und zwar als wir noch nicht richtig dividieren konnten. Wenn wir damals 5 / 2 ausgerechnet haben, dann hieß das Ergebnis: zwei Restwert eins. Oder bei 4 / 2 hieß das Ergebnis zwei Restwert null. Können wir uns noch dunkel erinnern?

Es stellt sich heraus, dass der Restwert Operator äußert nützlich ist, deswegen gibt es in Java dafür ein extra Zeichen, das Prozentzeichen: '%'. Eigentlich verwenden wir den Restwert Operator jeden Tag: wenn wir sagen es ist 2 Uhr nachmittags, dann haben wir implizit 14 % 12 im Kopf ausgerechnet.

Übung: Gerade Zahlen, ungerade Zahlen

Eine sinnvolle Übung ist es die Zahlen von 0 bis 6 aufzulisten und für jede dieser Zahlen den Restwert auszurechnen, also %2. Wir sehen, dass für gerade Zahlen der Restwert immer null ist und für ungerade Zahlen der Restwert immer eins. Das ist sehr nützlich, denn manchmal wollen wir wissen ob eine Zahl gerade oder ungerade ist.



Konstanten

Manche Variablen ändern sich nicht, sind also eigentlich Konstanten. Z.B die Kreiszahl 'Pi' hat immer den Wert '3.1415...'. Wir können das markieren mit dem Schlüsselwort *final*:

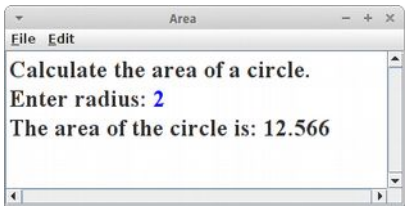
```
final double PI = 3.1415;
```

Konstanten können nachdem sie einmal initialisiert wurden nicht mehr verändert werden. Das mag am anfang etwas umständlich erscheinen, später werden wir aber sehen, dass die Verwendung von Konstanten zu viel besserem Code führt.

SEP: Konstanten sollten immer ganz in Großbuchstaben geschrieben werden, also z.B. 'MAX_NUM', damit man sie sofort von normalen Variablen unterscheiden kann.

Übung: Area

Schreiben Sie ein ConsoleProgram, das den Benutzer nach der Radius eines Kreises fragt, und dann dessen Fläche berechnet (Fläche = PI * Radius * Radius) und diese im Konsolenfenster ausgibt (println()). Dabei sollte PI als Konstante im Programm verwendet werden.



Boolesche Werte

Bisher haben wir uns mit den Zahlen, *int* und *double*, auseinandergesetzt. Jetzt wollen wir uns kurz den logischen Werten, auch *boolean* genannt, widmen. Um zu verstehen worum es geht betrachten wir die Ungleichung

```
3 > 5
```

also die Frage, ist drei größer als fünf. Die Antwort ist Nein. Man sagt auch die Aussage '3 > 5' ist falsch, *false* auf Englisch. Um damit umgehen zu können hat man den Datentyp *boolean* erfunden:

```
boolean b = 3 > 5;
```

d.h., also die Variable *b* ist vom Datentyp *boolean* und kann die Werte *true* oder *false* einnehmen. Das Ganze sollte uns etwas an Karel's Sensoren erinnern: z.B. der *beepersPresent()* Sensor hat immer *true* zurückgegeben, wenn ein Beeper da war und *false*, wenn keiner da war.

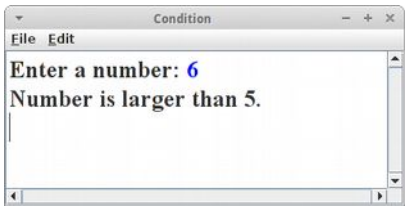
Bedingungen

Richtig Sinn machen boolesche Ausdrücke nur wenn sie in einer Bedingung verwendet werden. Bei Karel haben wir ja schon die *if* Anweisung kennen gelernt:

```
if ( beepersPresent() ) {
    pickBeeper();
} else {
    putBeeper();
}
```

Das Gleiche können wir jetzt auch mit einem Konsolenprogramm machen:

```
int x = readInt("Enter a number: ");
if ( x > 5 ) {
    println("Number is larger than 5.");
} else {
    println("Number is less than or equal to 5.");
}
```



Dass es sich bei 'x > 5' eigentlich um einen booleschen Ausdruck handelt, ist eigentlich eher nebensächlich.

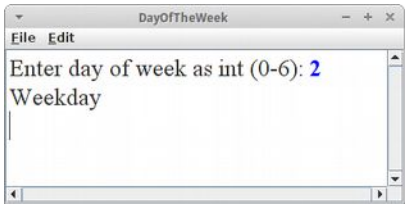
SEP: Wir sollten bei einer if Anweisung immer die geschweiften Klammern benutzen!

Vergleiche

Welche anderen Vergleiche außer '>' gibt es noch? Insgesamt sechs:

```
==      gleich
!=      nicht gleich
>       größer
<       kleiner
>=      größer oder gleich
<=      kleiner oder gleich
```

Übung: DayOfTheWeek



Schreiben wir ein Programm, das ausgibt, ob ein Tag ein Werktag, ein Samstag oder ein Sonntag ist:

```
int day =
    readInt("Enter day of week as int (0-6): ");
if (day == 0) {
    println("Sunday");
} else if (day <= 5) {
    println("Weekday");
} else {
    println("Saturday");

}
```

Falls also der Tag gleich null ist, dann ist es Sonntag; andernfalls, falls der Tag kleiner oder gleich 5 ist, dann ist es ein Werktag; andernfalls muss es der Samstag sein. Man bezeichnet diese Form der verketteten if Anweisung auch als 'cascading if'.

switch Anweisung

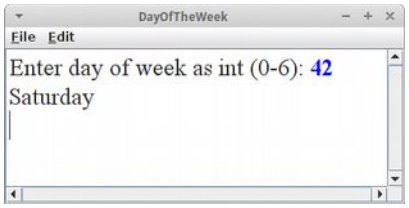
Die *switch* Anweisung gab es bei Karel noch nicht, aber sie stellt sich als sehr praktisch heraus. Im Prinzip macht sie das gleiche wie das 'cascading if', nur etwas eleganter:

```
int day = readInt("Enter day of week as int (0-6): ");
switch (day) {
case 0:
    println("Sunday");
    break;
case 6:
    println("Saturday");
    break;
default:
    println("Weekday");
    break;

}
```

Warum das eleganter sein soll wird sich etwas später zeigen.

SEP: Man sollte immer ein *default* bei der switch Anweisung haben, bzw. den letzten *else* Zweig beim cascading if.

Übung: DayOfTheWeek



Wir wollen unser DayOfTheWeek Problem jetzt mit der *switch* Anweisung lösen. Zunächst verwenden wir den Code wie oben, uns testen ob er auch funktioniert. Testen heißt, dass wir jede möglich Eingabe testen. (Was passiert wenn wir -1 oder 42 eingeben?)

Frage: Was passiert wenn wir eine der *break* Anweisungen weglassen?

Boolesche Operatoren

Erinnern wir uns an LumberjackKarel: im letzten Schritt, wenn Karel alle Bonbons einsammeln soll um sie auf einen großen Haufen zu legen, wäre es toll gewesen wenn wir zwei Bedingungen gleichzeitig hätten testen können, also ist links frei *und* ist ein Bonbon da:

```
while ( leftIsBlocked() && beepersPresent() ) {...}
```

Hier bedeutet das '&&' soviel wie 'und', also nur wenn beide Bedingungen erfüllt sind, soll Karel etwas machen. Ähnlich praktisch wäre es gewesen wenn wir Karel nur dann etwas hätten machen lassen wenn etwas *nicht* erfüllt ist,

```
if ( !beepersPresent() ) {...}
```

Hier bedeutet das '!' soviel wie 'nicht'. Schließlich, hatten wir das Beispiel von AriadneKarel wo Karel etwas tun sollte, wenn entweder vor ihm eine Wand war *oder* keine Bonbons da war:

```
if ( frontIsBlocked() || noBeepersPresent() ) {...}
```

Das '||' bedeutet 'oder' im Sinne von entweder-oder.

Diese drei Operatoren sind die sogenannten booleschen Operatoren:

```
!       not
&&      and
||      or
```

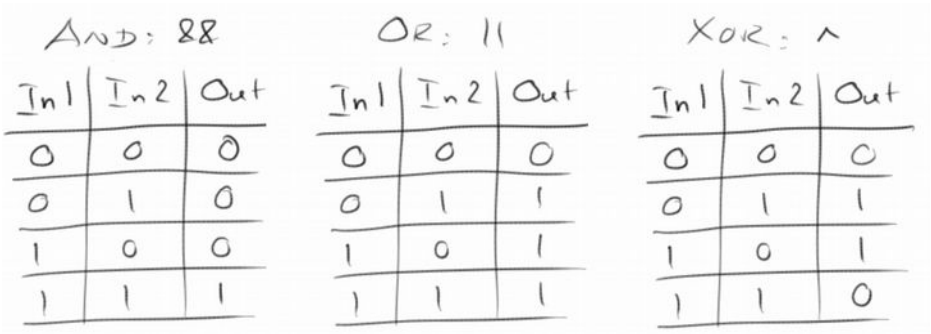
Die Reihenfolge spiegelt auch die Vorrangsregeln wider, also '!' hat eine höhere Priorität als 'and'.

Wahrheitstabellen

Es ist üblich die booleschen Operationen in sogenannten Wahrheitstabellen (truth tables) darzustellen. Dabei sind *in1* und *in2* die zwei Operanden, und *out* ist das Resultat, z.B.

```
boolean out = in1 && in2;
```

für die '*and*' Operation. Die Wahrheitstabellen für die drei logischen Operationen sehen wie folgt aus:

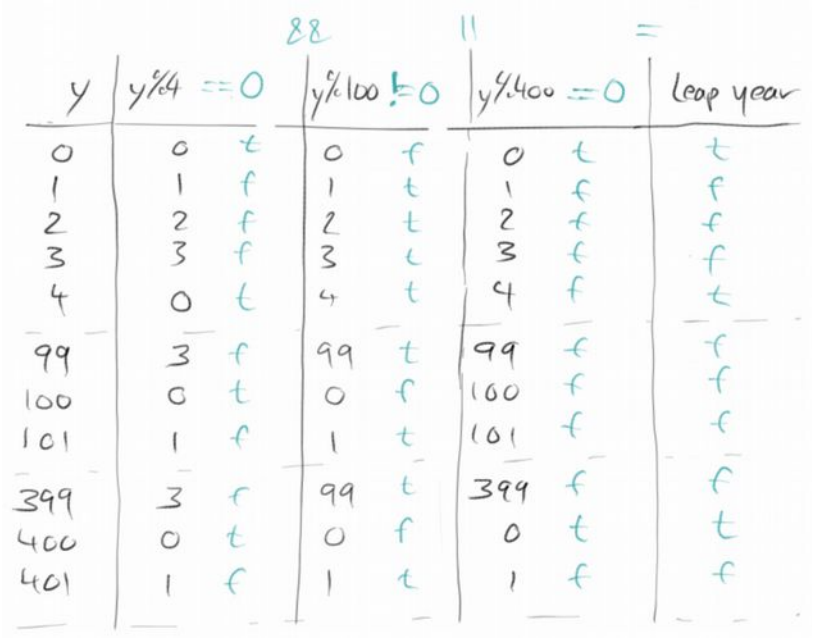


Übung: LeapYear

Es gibt einen coolen booleschen Ausdruck der uns angibt, ob ein Jahr ein Schaltjahr ist oder nicht:

```
boolean p = ((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0);
```

Um den Ausdruck zu verstehen, schreibt man sich am besten die Wahrheitstabellen (truth table) dafür auf.



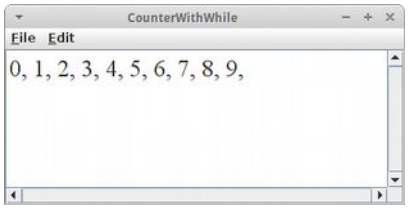
y	y%4 == 0		y%100 != 0		y%400 == 0		leap year
0	0	t	0	f	0	t	t
1	1	f	1	t	1	f	f
2	2	f	2	t	2	f	f
3	3	f	3	t	3	f	f
4	0	t	4	t	4	f	t
99	3	f	99	t	99	f	f
100	0	t	0	f	100	f	f
101	1	f	1	t	101	f	f
399	3	f	99	t	399	f	f
400	0	t	0	f	0	t	t
401	1	f	1	t	1	f	f

while Schleife

Wir haben die *while* Schleife schon bei Karel kennengelernt:

```
while ( frontIsClear() ) {
    move();
}
```

Die *while* Schleife wird solange ausgeführt, solange eine bestimmte Bedingung erfüllt ist. Als einfach Beispiel wollen wir die Zahlen von 0 bis 9 ausgeben.




```
int i = 0;
while ( i < 10 ) {
    println( i );
    i = i + 1;
}
```

In der ersten Zeile deklarieren wir eine Variable namens *i* von Datentyp *int* auf den Wert 0. Dann testen wir ob *i* kleiner als 10 ist. Solange dies der Fall ist, drucken wir den momentanen Wert von *i* im Konsolenfenster, und dann erhöhen wir den Wert von *i* um eins.

Diese letzte Zeile mag etwas ungewöhnlich sein (speziell Mathematiker haben ein Problem damit). Aber es ist wichtig sich daran zu erinnern, dass '=' nicht für Gleichheit steht, sondern für *Zuweisung*. Also

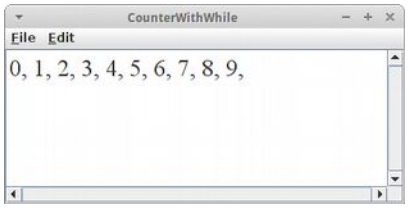
```
i = i + 1;
```

bedeutet soviel wie: nimm den momentanen Wert von *i*, addiere dazu eins, und weise das Resultat dann der Variablen *i* zu.

Übung: CounterWithWhile

Als kleine Übung probieren wir den Zählercode aus: einfach die Zeilen oben in die run() Methode eines ConsolePrograms einfügen. Was passiert, wenn wir anstelle von *println()* nur *print()* verwenden?

for Schleife



Wir haben die *for* Schleife ja schon ganz am Anfang bei Karel kennengelernt. Damals hat uns nur die Zahl, wie oft die Schleife durchlaufen wird, interessiert. Jetzt sind wir aber soweit den Rest auch zu verstehen. Die folgende *for* Schleife gibt auch die Zahlen von 0 bis 9 aus:

```
for ( int i = 0; i < 10; i++ ) {
    println(i);
}
```

Im Allgemeinen sieht eine *for* Schleife immer wie folgt aus:

```
for ( init; condition; step ) {
    statements;
}
```

Als erstes wird der *init* Schritt ausgeführt, meist so etwas wie 'int i = 0'. Dann wird die *condition* gecheckt, also ist 'i < 10'? Falls ja werden die *statements* innerhalb der Schleife ausgeführt. Zum Schluß wird dann *step* ausgeführt, also 'i++'. Das wird solange getan bis die *condition* nicht mehr erfüllt ist.

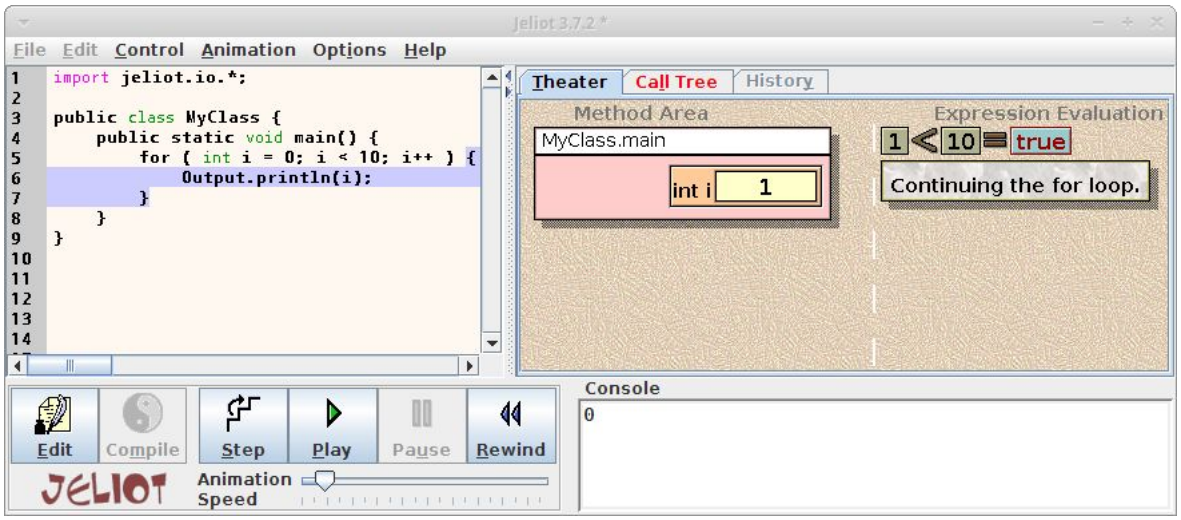
Was bedeutet dieses 'i++'? Man nennt es den Inkrement Operator, und er bedeutet soviel wie die Zeile

```
i = i + 1;
```

also der Wert der Variablen *i* wird um eins erhöht. Es gibt auch einen Dekrement Operator der genau das Gegenteil macht: 'i--'.

Übung: for Schleife in Jeliot

Um die *for* Schleife noch besser zu verstehen, sehen wir uns die Schleife mal in Jeliot an, und beobachten Schritt für Schritt was passiert.



for versus while

Wie wir gesehen haben, machen *while* Schleife und *for* Schleife eigentlich das Gleiche. Daher stellt sich die Frage wann verwendet man welche?

- **for:** verwenden wir, wenn wir von vornherein wissen wie oft etwas ausgeführt wird, z.B., `for ( int i=0; i<10; i++ ) {...}`
- **while:** verwenden wir wenn wir noch nicht genau wissen wie oft eine Schleife ausgeführt wird, z.B.,
  `while ( frontIsClear() ) {...}`

OBOB: FillRowKarel

Es gibt noch eine Schleife über die wir noch nicht gesprochen haben, den 'Loop and a Half': Er ist die Lösung unseres OBOB Problems. Erinnern wir uns an FillRowKarel:

```
while ( frontIsClear() ) {
    putBeeper();
    move();
}
putBeeper();
```

Das Problem liegt darin, dass wir das 'putBeeper()' Kommando zweimal benötigen. Diese Duplizierung von Code tritt bei jedem OBOB auf und doppelter Code ist immer etwas Schlechtes.

SEP: Vermeide doppelten Code.

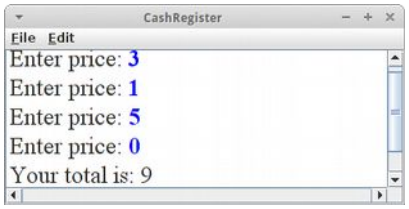
Loop and a Half

Die Lösung ist eigentlich ganz einfach und heißt 'Loop and a Half':

```
while ( true ) {
    putBeeper();
    if ( frontIsBlocked() ) break;
    move();
}
```

Analysieren wir den Code: Zunächst einmal haben wir eine Endlosschleife, 'while (true)'. Allerdings können wir mit der Anweisung *break* diese Endlosschleife beenden, also aus ihr herausspringen. Das machen wir dann wenn vor Karel eine Wand ist, 'if (frontIsBlocked())'. Jetzt sehen wir auch warum es 'Loop and a Half' heißt: die Schleife wird im letzten Schritt nur zur Hälfte ausgeführt. Um den Code zu verstehen, sollten wir ihn Schritt für Schritt auf einem Stück Papier durchgehen.

Übung: CashRegister



Eine schöne und typische Anwendung für den 'Loop and a Half' ist die Kasse in einem Laden: wir haben z.B. fünf Sachen in unserem Einkaufswagen und wollen jetzt bezahlen. Der Kassierer tippt also die Preise der einzelnen Waren, eine nach der anderen in die Kasse, die diese addiert und am Ende gibt sie aus wie viel wir bezahlen sollen:

```
int total = 0;

while (true) {
    int price = readInt("Enter price: ");
    if ( price == 0 ) break;
    total += price;
}
println("Your total is: " + total);
```

Da aber nicht alle Leute immer genau fünf Sachen kaufen, brauchen wir irgendein Abbruchkriterium: in unserem Fall ist es wenn der Kassierer die 0 für den Preis eingibt. Man nennt dieses Abbruchkriterium im Englischen manchmal auch den Sentinel.

Frage: Könnten wir auch eine negative Zahl als Abbruchkriterium verwenden?

SEP: Man sollte möglichst nicht mehrere 'breaks' verwenden.

Post-Increment vs Pre-Increment

Wir haben bisher nur den Inkrement Operator als Post-Inkement Operator gesehen, es gibt ihn aber auch als Pre-Inkement. Der Unterschied ist, dass einmal das '++' vor der Variablen steht (pre) oder danach (post). Also:

```
int x = 5;
int y = x++;    // Post: y=5
```

oder

```
int x = 5;
int y = ++x;    // Pre: y=6
```

Der Unterschied ist subtil, und man bemerkt ihn nur wenn dieser Operator in Zusammenhang mit Assignments oder anderen Operationen steht:

```
int a = 6;
int x = ++a;
int y = x++;
```

In der zweiten Zeile wird zuerst (pre) die Variable *a* um eins erhöht, und danach die Zuweisung gemacht. In der letzten Zeile, wird zuerst die Zuweisung gemacht, und danach (post) die Variable *x* um eins erhöht.

Review

Was haben es geschafft! Dieses Kapitel war etwas schwieriger als die beiden vorhergehenden. Dafür geht es von nun an aber bergab. Was haben wir in diesem Kapitel gelernt? Wir haben

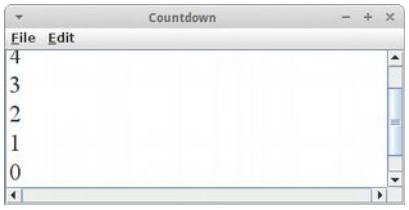
- Variablen kennengelernt,
- uns an den Restwert Operator erinnert,
- unseren ersten Kontakt mit Konstanten gehabt,
- Boolesche Operatoren und Wahrheitstabellen zum ersten Mal gesehen,
- festgestellt, dass die *if* und die *switch* Anweisung sehr ähnlich sind,
- gelernt wann wir eine *for* und wann eine *while* Schleife verwenden sollen und
- im 'Loop and a Half' die Lösung für unser OBOB Problem gefunden.

Das wichtigste in diesem Kapitel war, dass wir unsere ersten Schritte auf eine solide Basis gestellt haben.

Projekte

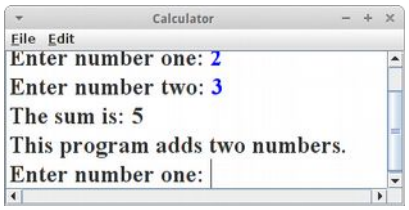
Die folgenden Projekte sind vielleicht nicht ganz so interessant wie die des letzten Kapitels. Aber sie stellen die Grundlage für die folgenden Kapitel dar.

Countdown



Karel liebt Raketen und besonders den Countdown kurz vorher. Deswegen wollen wir ein Programm schreiben, dass auf der Konsole die Zahlen von 10 bis 0 ausgibt.

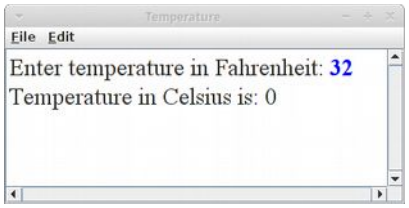
Calculator



Das mit dem DoubleBeeper war für Karel ziemlich viel Arbeit. Mit Konsolenprogrammen und Variablen geht das viel einfacher. Deswegen wollen wir ein Konsolenprogrammen schreiben, das zwei Zahlen addiert.

Was etwas störend an dem Programm ist, das man es jedes mal neu starten muss, wenn man eine neue Addition machen will. Was könnte man tun, damit das Programm mehrere Additionen (immer von zwei Zahlen) erlaubt?

Temperature



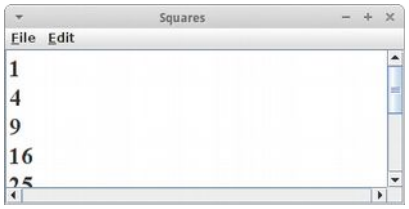
Karel liebt Amerika, aber er kann nichts mit diesen Fahrenheit anfangen. Deswegen schreiben wir ihm ein Programm, das Fahrenheit in Celsius umrechnet. Wir bitten den Benutzer uns eine Temperatur in Fahrenheit zu geben. Mittels readInt() speichern wir diese in der Variablen *f*. Und mit der Formel

```
int c = (int) ( (5.0 / 9.0) * (f - 32) );
```

können wir das in Celsius umrechnen.

Neu an dieser Formel ist was das `(int)` macht: Es handelt sich hier um einen *Cast* oder Typumwandlung auf Deutsch. Dazu müssen wir kurz überlegen um welchen Datentype es sich denn bei `(5.0 / 9.0) * (f - 32)` handelt? Sobald einer der Operanden ein *double* ist, wird das ganze Ergebnis in einen *double* umgewandelt. Dann haben wir aber auf der einen Seite der Zuweisung einen *double* stehen und auf der anderen aber eine *int*. Das ist nicht gut. Deswegen konvertieren wir den *double* in einen *int*, und genau das macht `(int)`.

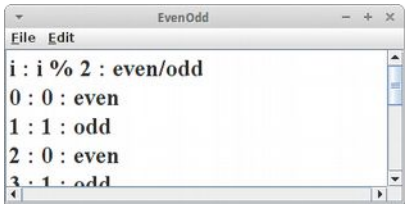
Squares



Als Beispiel für die Verwendung von Konstanten wollen wir ein ConsoleProgram schreiben, das die Quadratzahlen der Zahlen von 0 bis MAX_NUM ausgibt, wo MAX_NUM eine Konstante sein soll die auf den Wert 10 gesetzt wird.

SEP: Zauberzahlen (magic numbers): Zauberzahlen sind Zahlen die irgendwo in unserem Code auftauchen, und man hat meist keine Ahnung wo sie herkommen und was sie bedeuten. Deswegen sollten alle Zahlen außer die 0, 1 und 2 als Konstanten deklariert werden.

EvenOdd



Wir wollen ein kleines ConsoleProgram schreiben, dass die Zahlen von 0 bis 10 listet, daneben den Wert der Zahl % 2, und schließlich ob die Zahl gerade oder ungerade ist.

Money

Wenn wir in unseren Programmen mit Geld zu tun haben stellt sich zunächst die Frage sollen wir dafür einen *int* Datentyp oder einen *double* Datentyp verwenden? Die Antwort ist ganz einfach, wenn man sich die Frage stellt: kann man Geld zählen? Für alles was man zählen kann verwendet man den int Datentyp.

Das heißt wenn wir mit Geld arbeiten sollten wir immer mit Cents rechnen. Allerdings wenn wir diese dann ausgeben, wäre es besser wenn wir nicht 120 Cents ausgeben würden sondern 1,20 Euro. Auch hier ist der Restwert Operator '%' von großem Nutzen:

```
int money = 120;
int euros = money / 100;
int cents = money % 100;
println("The amount is " + euros + "," + cents + " Euro.");
```

Um sicher zu stellen, dass unser Code wirklich funktioniert, sollten wir verschiedene Eingaben ausprobieren. Gute Testkandidaten sind die Eingaben: 120, 90, 100, 102 und 002. Hat unser Programm immer die richtige Ausgabe zurückgegeben?

BigMoney

Bei Beträgen über tausend Euro wird noch bei den Tausendern und auch bei den Millionen noch ein Punkt eingefügt, also z.B.: 1.001.233,45 Euro. Wir wollen also eine Methode *formatNumericString(int cent)* schreiben, die einen Geldbetrag in Cent als Übergabeparameter erhält, und einen String zurückliefert, der richtig formatiert ist. Wenn wir das Programm testen, sollten wir den obigen Betrag verwenden. Wir werden feststellen, dass es Sinn macht eine Methode *padWithZeros()* zu schreiben.

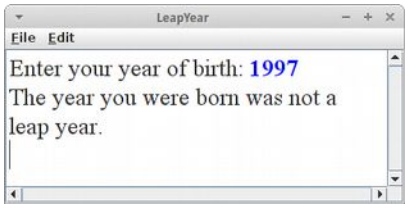
Auch hier sollten wir verschiedene Eingaben testen, z.B. 100123345, 100123305, und 05.

Time

Menschen bevorzugen es die Uhrzeit in Stunden, Minuten und Sekunden anzugeben. Computer hingegen denken nur in Sekunden, also Sekunden die seit Mitternacht vergangen sind. Deswegen wollen wir ein Programm schreiben, das aus den Sekunden die seit Mitternacht vergangen sind, die momentane Uhrzeit berechnet. Dazu benötigen wir wieder den Restwert Operator. Unter Umständen macht es auch wieder Sinn eine Methode *padWithZeros()* zu schreiben, die sicher stellt, dass anstelle von "6" Minuten "06" Minuten angezeigt werden.

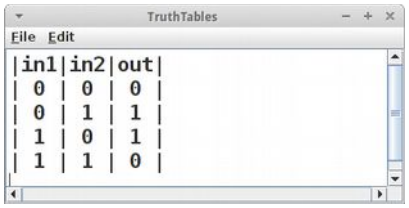
Zum Testen sollten wir mindestens die folgenden Eingaben ausprobieren: 5, 61, 85, 3600, 3601.

LeapYear



Wir haben ja oben gesehen wie man feststellt ob ein Jahr ein Schaltjahr ist. Mit diesem Wissen wollen wir ein Konsolenprogramm schreiben, das ausgibt, ob das Jahr in dem jemand geboren wurde ein Schaltjahr war. Z.B., sollte 1996, 2000, und 2004 ein Schaltjahr sein, 1900 und 2000 sollten aber keine sein.

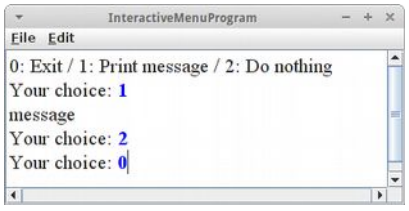
TruthTables*



In diesem Projekt wollen wir uns mit dem booleschen Datentyp und den logischen Operatoren && (und), || (oder) und ^ (exklusives oder) auseinandersetzen. Wir wollen dafür ein Programm schreiben, dass die Wahrheitstabellen für alle drei Operatoren berechnet und ausgibt. Evtl. hilfreich ist der folgende Trick wie man aus einem *int* Datentyp einen *boolean* Datentyp erzeugt:

```
int in1 = 0;
boolean b1 = (in1 != 0);
```

InteractiveMenuProgram



Sehr häufig benötigt man in einem Konsolenprogramm eine Art Menu. Der Benutzer kann über die Eingabe einer Zahl zwischen verschiedenen Menupunkten auswählen. Wir wollen daher ein Programm schreiben, in dem der Benutzer zwischen drei Möglichkeiten auswählen kann: wenn er '0' eingibt, soll das Programm beendet werden, wenn er '1' eingibt soll eine Nachricht ausgegeben werden, und wenn er '2' eingibt soll gar nichts passieren. Als erste soll ihm natürlich mitgeteilt werden was seine Optionen sind:

```
println("0: Exit / 1: Print message / 2: Do nothing");
```

Dann beginnt ein 'Loop and a Half', also `while (true)`. Wir fragen den Benutzer nach seiner Wahl

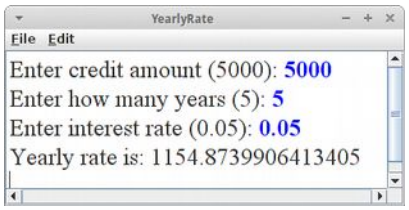
```
int choice = readInt("Your choice: ");
```

Wenn der Benutzer die '0' eingibt beenden wir den 'Loop and a Half'

```
if (choice == 0) break;
```

Danach folgt dann ein *switch* oder *cascading if* für die Wahlmöglichkeiten '1' und '2'.

YearlyRate



Karel will sich ein Auto kaufen, deswegen hat er angefangen zu sparen. Er will sich einen Mini kaufen (Mercedes ist zu teuer), und er hat einen gebrauchten für 5000 Euro gesehen. Er hat einen günstigen Kredit bei einer Bank von 5% pro Jahr gesehen. Wie hoch ist seine jährliche Rate, wenn er den Kredit in 5 Jahren abbezahlt haben will?

Karel braucht also ein Programm bei dem er die Kreditsumme (k) eingeben kann, den Zinssatz (z) und die Laufzeit in Jahren (n). Das Programm soll ihm dann sagen wie hoch seine jährliche Rate (y) ist. Die Formel dafür kann man aus der Wikipedia unter Sparkassenformel nachsehen [2]:

```
double q = 1.0 + z;
double qn = Math.pow(q, n);    // q^n
double y = k * qn * (q-1) / (qn-1);
```

Hilfreich zu wissen ist noch, dass man Gleitkommazahlen mit readDouble() einlesen kann.

Fragen

1. Nennen Sie vier der primitiven Datentypen von Java.

2. Was ist eine Zauberzahl (magic number)?

3. Was ist der Unterschied zwischen einem GraphicsProgram und einem ConsoleProgram?

4. In der Vorlesung haben Sie von vielen Software Engineering Prinzipien gehört. Geben Sie drei Beispiele.

5. Beim Programmieren gibt es einige typische Fehler die immer wieder passieren (Common Errors). Nennen Sie zwei dieser Fehler.

6. Schreiben Sie ein ConsoleProgram, das den Benutzer nach der Radius eines Kreises fragt, und dann dessen Fläche berechnet (Fläche = PI * Radius * Radius) und diese auf der Console ausgibt.

7. Was ist der Unterschied zwischen 'print("hi")' und 'println("hi")'?

8. Was macht die 'readInt()' Methode?

9. Was sind die Werte der Variablen 'x' und 'y' nachdem die folgenden drei Zeilen ausgeführt wurden?
```
a = 6;
x = ++a;
y = x++;
```

10. Was ist das Ergebnis der folgenden Ausdrücke:
```
- int m = (int) 3.7688;
- int n = 14 % 12;
- int o = 4 + 5 * 3 / 2;
```

11. Was ist das Resultat des folgenden Ausdrucks?
```
int x = 3 - 3 * 6 / 4;
```

12. Was ist der Unterschied zwischen
    a) 13 / 5
    b) 13 / 5.0

13. Viele Programmierfehler rühren von fehlenden Klammern und/oder Leerzeichen. Die folgenden Beispiele sollen verdeutlichen, dass es einfacher ist Klammern zu setzen als die Operatorvorrangsregeln (rules of precedence) auswendig zu lernen.

    i) Was ist der Wert von 'j'?
    ```
    int i = 5;
    int j = i+++--i;
    ```

    ii) Was ist der Wert von 'y'?
    ```
    int y = 3 * 4 % 5 * 6 - 3;
    ```

    iii) Was ist der Wert von 'b'?
    ```
    int x = 5;
    int z = 3;
    boolean b = x != 5 || x < z == z < 3;
    ```

14. Angenommen, Sie haben eine Varable x vom Typ int gegeben, also int x = 3;. Wie kann man diese in eine Variable namens y vom Typ double umwandeln (cast)?

15. Beschreiben Sie mit Ihren eigenen Worten was der folgende Ausdruck bedeutet:
    ```
    (x < 0) || (x > WIDTH)
    ```

16. Was ist der Vorteil wenn man Konstanten verwendet?

17. Vergleichen Sie die beiden 'switch' Beispiele unten, und beschreiben Sie ausgegeben wird.
    a)
    ```
    int day = 0;
    switch (day) {
    case 0:
        println("Sunday");
        break;
    case 6:
        println("Saturday");
        break;
    }
    ```

    b)
    ```
    int day = 0;
    switch (day) {
    case 0:
        println("Sunday");
    case 6:
        println("Saturday");
    }
    ```

18. Welche Werte kann eine boolesche (boolean) Variable annehmen?

19. Welchen Datentyp muss die Variable 'b' in dem Beispiel unten haben?
    ```
    if ( b ) {
        println("hi");
    }
    ```

20. Schreiben Sie den Test der feststellt, ob eine Zahl y ist durch 400 teilbar ist.

21. Erinnern Sie sich an den Remainder ’%’ Operator? Geben Sie zwei Beispiele wofür man diesen gebrauchen kann.

22. Menschen bevorzugen es Zeiten in Stunden, Minuten und Sekunden anzugeben. Computer bevorzugen es nur in Sekunden zu rechnen. So ist z.B. 13 Uhr 35 Minuten und 12 Sekunden für den Computer einfach 48912 Sekunden. Benutzen Sie dieses Wissen um die Computerzeit 't' in eine menschenlesbare Form, also Stunden, Minuten und Sekunden umzuwandeln. Sie dürfen auch mehrere Zwischenschritte verwenden.

23. Wann sollte man eine for–Schleife und wann eine while–Schleife benutzen?

24. Alles was man mit einer 'for' Schleife machen kann, kann man auch mit einer 'while' Schleife machen und umgekehrt. Schreiben Sie die folgende 'for' Schleife als 'while' Schleife um.

```
for (int i=0; i<5; i++) {
       println(i);
   }
```

25. Beschreiben Sie den Aufbau des 'loop-and-a-half'. Welches Prolem löst dieser?

26. Loop and a Half (Schleife und eine Halbe): Schreiben Sie das folgende Karel Beispiel mithilfe des Loop and a Half. Was ist der Vorteil des Loop and a Half?

```
public void run() {
 while ( frontIsClear() ) {
 putBeeper();
 }
 putBeeper();
 }
```

Referenzen

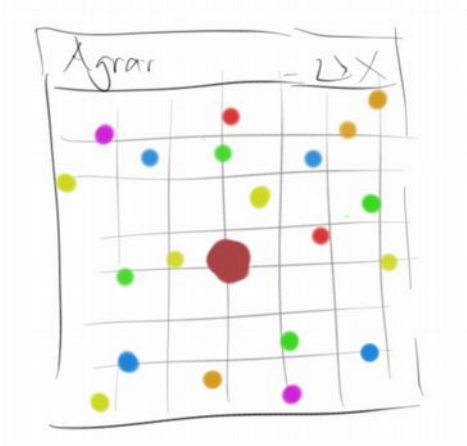
Referenzen zu den Themen in diesem Kapitel finden sich in praktisch jedem Buch zu Java, insbesondere natürlich auch in [3].

[1] Jeliot 3, Program Visualization application, University of Helsinki, cs.joensuu.fi/jeliot/description.php

[2] Seite „Sparkassenformel“. In: Wikipedia, Die freie Enzyklopädie. URL: https://de.wikipedia.org/w/index.php?title=Sparkassenformel&oldid=143971427

[3] The Art and Science of Java, von Eric Roberts, Addison-Wesley, 2008

Agrar



In diesem Kapitel fängt es an richtig interessant zu werden. Im letzten Kapitel haben wir den Top-Down Ansatz auf Kosten von Variablen etwas vernachlässigt, aber in diesem Kapitel werden wir sehen wie wir mit Methoden wieder zum Top-Down Ansatz zurückfinden. Und wir werden unsere ersten Animationen für Spiele schreiben.

Methoden

Wir kennen Methoden schon relativ lange, bei Karel haben wir sie noch Kommandos genannt, z.B. *move()*, *turnLeft()* und *frontIsClear()*. Auch in GraphicsPrograms haben wir sie benutzt, damals hießen sie auch Nachrichten die wir an ein GRect schicken, z.B. *setColor()* und *setFilled()*. Selbst bei ConsolePrograms hatten wir sie, z.B. *readInt()* und *println()*. Also nix neues.

Bei Karel gab es aber die coole Möglichkeit neue Kommandos zu erfinden, wie z.B. *turnRight()* oder *moveToWall()*. Das haben wir damals so gemacht:

```
public void turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
}
```

Wäre es nicht auch cool gewesen, wenn wir in unserem Target Programm folgendes neues Kommando hätten haben können:

```
public void drawCircle( int radius, Color color ) {
    ...
}
```

oder für unser Wall Programm wäre eine Methode wie die folgende ganz praktisch gewesen:

```
public void drawRowOfNBricks( int numberOfBricks ) {
    ...
}
```

Nun das schöne ist, das geht tatsächlich!

SEP: Methoden tun immer etwas, deswegen sollten Methoden immer Tunwörter, also Verben, sein.

Neue Methoden definieren

Eine neue Methode anzulegen ist genauso einfach wie Karel neue Kommandos beizubringen. Der allgemeine Syntax einer Methodendeklaration ist folgender:

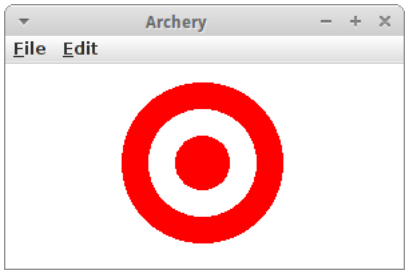
```
visibility type name( parameters ) {
    ... body...

}
```

Dabei bedeutet

- **visibility:** die Sichtbarkeit der Methode, dies ist meist entweder *private* oder *public*
- **type:** der Datentyp den die Methode zurückgibt, auch Rückgabewert genannt. Sehr häufig ist das *void*, was soviel heißt wie nichts, also die Methode gibt nichts zurück
- **name:** der Name der Methode, hier gelten die gleichen Regeln wie für Variablen und er sollte auch immer klein geschrieben werden
- **parameters:** die Übergabe Parameter. Die sind neu, die gab es bei Karel noch nicht, die sind aber super praktisch wie wir gleich sehen werden

SEP: Wenn möglich sollte die Sichtbarkeit von Methoden *private* sein.

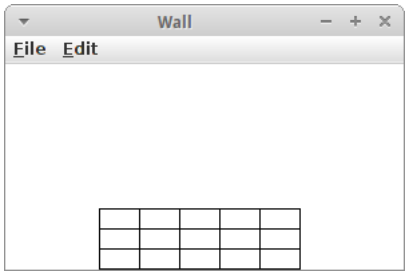
Übung: Archery



Schauen wir uns unser Archery Programm noch einmal an. Aber jetzt versuchen wir es mit Hilfe der Methode *drawCircle(int radius, Color color)*. Wir werden feststellen, dass der Code viel kürzer und auch lesbarer wird.

```
private void drawCircle(int radius, Color color) {
    GOval ring =
        new GOval(2 * radius, 2 * radius);
    ring.setColor(color);
    ring.setFilled(true);
    int x = 75 + 72 - radius;
    add(ring, x, x);
}
```

Übung: Wall



Versuchen wir uns noch einmal an dem *Wall* Programm, aber jetzt mit Hilfe von Methoden. Wir wollen also eine Mauer bestehend aus 15 Backsteinen (GRect) errichten. Dabei wollen wir aber die Methode

```
private void drawOneRowOfStones(int y) {
    int x = 50;    // beginning x position of wall
    for (int i = 0; i < 5; i++) {
        GRect brick = new GRect(30, 15);
        add(brick, x, y);
        x = x + 30;
    }
}
```

verwenden.

Frage: Enthält die Methode Zauberzahlen (magic numbers)? Können wir das ändern?

Rückgabewert

Bisher habe wir nur Methoden gesehen, die nichts (void) zurückgeben. Es gibt aber auch Methoden die etwas zurückgeben. Meistens rechnen die Methoden etwas aus, und geben das Resultat als Rückgabewert zurück. Ein schönes Beispiel ist die folgende Methode die zurückgibt wie viel Inches in einer gegebenen Anzahl von Feet sind:

```
private double feetToInches(double feet) {
    double inches = 12 * feet;
    return inches;
}
```

Wir können diese Methode jetzt in einem ConsoleProgram verwenden.

Übung: ConvertFeetToInches

Wir wollen ein kleine ConsoleProgram schreiben, das Feet nach Inches konvertiert. Dazu fragt es den Benutzer nach einer Anzahl von Feet mittels *readDouble()* und ruft dann unsere feetToInches() Methode auf und gibt das Resultat im KonsolenFenster aus.

```
double feet = readDouble("Enter feet: ");
double inches = feetToInches(feet);
println(feet + " feet are " + inches + " inches.");
```

Wenn wir genau hinschauen, sehen wir, dass wir die feetToInches() Methode genauso verwenden wie die readInt() oder println() Methoden. Der einzige Unterschied ist, dass wir sie selbst geschrieben haben.

Objekte als Rückgabewert

Wir können nicht nur Zahlen als Rückgabewerte haben, sondern jeden möglichen Datentyp, also auch GRects zum Beispiel. Die folgende Methode generiert einen farbig ausgefüllte Kreis mit vorgegebenem Radius *r* an der Position *x* und *y*:

```
private GOval createFilledCircle(int x, int y, int r, Color color) {
    GOval circle = new GOval(x-r, y-r, 2*r, 2*r);
    circle.setFilled(true);
    circle.setColor(color);
    return circle;
}
```

Wir müssen in jetzt lediglich zu unserem GraphicsProgram hinzufügen.

Lokale Variablen

Die Variablen die wir bisher kennengelernt haben nennen wir auch *lokale* Variablen. Lokal im Bezug auf eine Methode. Das soll heißen, dass Variablen nur in der Methode sichtbar sind in der sie deklariert wurden, außerhalb, also in anderen Methoden sind sie nicht sichtbar. Betrachten wir das an einem Beispiel:

```
public class LocalVariables extends ConsoleProgram {

    public void run() {
        double feet = readDouble("Enter feet: ");
        println(feet);
    }

    private void feetToInches() {
        double inches = 12;
        println(inches);
    }
}
```

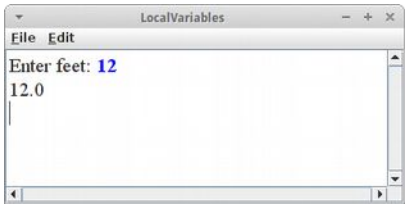
In dem Beispiel gibt es eine Variable *feet* die in der Methode *run()* existiert und eine Variable *inches* die in der Methode *feetToInches()* existiert. Wenn wir versuchen in der Methode *feetToInches()* auf die Variable feet zuzugreifen, dann geht das nicht. Umgekehrt gilt das gleiche. Die Variablen sind also nur lokal in ihren jeweiligen Methoden sichtbar.

Wie wir aber gesehen haben ist es aber viel besser wenn wir von einer Methode auf die Variablen einer anderen Methode zugreifen können. Deswegen benötigen wir Parameter.

```
private void feetToInches(double feet) {
    double inches = 12 * feet;
    println(inches);
}
```

Mit den Parametern übergeben wir Variablen von einer Methode an eine andere, deswegen nennt man sie auch Übergabeparameter.

Allerdings wird nur eine Kopie der Variablen übergeben. Um das zu sehen, schauen wir uns das folgende Programm kurz an:




```
public class LocalVariables extends ConsoleProgram {

    public void run() {
        double feet = readDouble("Enter feet: ");
        feetToInches(feet);
        println(feet);
    }

    private void feetToInches(double feet) {
        feet = 42;
    }
}
```

Erst bitten wir den Benutzer uns einen Wert zu geben, z.B. '12'. Diesen Wert übergeben wir an die Methode *feetToInches()*. Wenn wir dann aber sagen, dass `feet = 42` sein soll, dann gilt das aber nur für die Kopie, das Original bleibt unversehrt, wie wir sehen wenn wir uns anschauen was im Konsolenfenster ausgegeben wird. Dort erscheint nämlich die '12'.

Deswegen ist es eigentlich auch egal, ob wir das rote *feet* auch *feet* nennen oder ihm einen anderen Namen geben:

```
public class LocalVariables extends ConsoleProgram {

    public void run() {
        double feet = readDouble("Enter feet: ");
        feetToInches(feet);
        println(feet);
    }

    private void feetToInches(double fritz) {
        fritz = 42;
    }
}
```

Jetzt wird hoffentlich auch klar, warum wir einen Rückgabewert benötigen. Denn wenn in der aufgerufenen Methode *feetToInches()* irgendetwas ausgerechnet wird, dann ist das nur lokal in der Methode sichtbar. Damit wir es wieder zurück in die aufrufende Methode bekommen, benötigen wir den Rückgabewert.

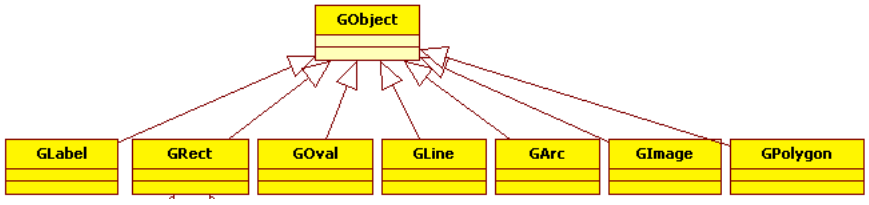
Eine Anmerkung noch: eine Methode kann mehrere Parameter haben, sie kann aber nur einen Rückgabewert haben.

Animation

So, jetzt haben wir Methoden in unserer Toolbox. Das ist super, denn damit können wir anfangen wirklich coole Sachen zu machen, nämlich Animationen und Spiele. Wir fangen mit Animationen an. Dafür brauchen wir aber noch eine Sache.

GObject

Erinnern wir uns an die Grafik Klassen die wir im zweiten Kapitel kennengelernt haben: GRect, GOval, GLine, GImage, GLabel, GArc und GPolygon. Diese Klassen sind nicht ganz unabhängig voneinandern, sondern sie haben sogar etwas gemeinsam, sie sind nämlich alle GObjects.

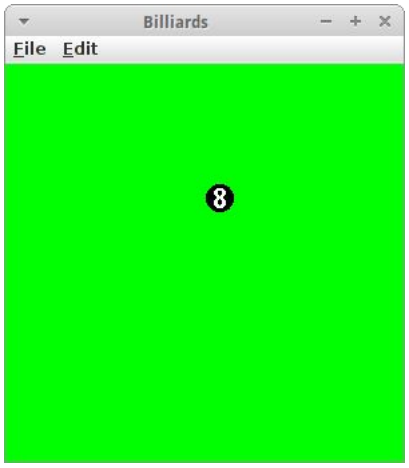


Die Klasse GObject nennt man auch Elternklasse. Und man sagt auch die Kinderklassen wie GRect und GOval erben von ihrer Elternklasse. Was erben sie? Die Methoden der Elternklasse. Das ist super praktisch wie wir sehen werden. Aber erst einmal schauen wir uns an welche Methoden das sind:

- **setLocation(x, y):** schiebt ein GObject an die Stelle x, y
- **move(dx, dy):** verschiebt ein GObject um dx und dy
- **getX():** gibt die x Koordinate des GObjects
- **getY():** gibt die y Koordinate des GObjects
- **getWidth():** gibt die Breite des GObjects
- **getHeight():** gibt die Höhe des GObjects
- **setColor(col):** ändert die Farbe des GObjects
- **sendToFront():** schickt eine GObject nach vorne (z-order)
- **sendToBack():** schickt eine GObject nach hinten (z-order)

Also alle Grafik Klassen die wir bisher kennengelernt haben, haben diese Methoden.

Übung: Animationen mit dem Game Loop



Für unsere erste Animation haben wir uns Billiard ausgesucht: Wir wollen, dass sich eine scharze Kugel über den Tisch bewegt, und an den Seiten reflektiert wird.

Jede Animation hat einen "*Game Loop*". Unsere *run()* Methode sieht wie folgt aus:

```
public void run() {
    setup();

    // game loop
    while (true) {
        moveBall();
        checkForCollisionsWithWall();
        pause(40);
    }
}
```

Alles was mit dem Setup und der Initialisierung zu tun hat, kommt in die *setup()* Methode. Danach beginnt der *Game Loop*: das ist effektiv eine Endlosschleife. In dieser Endlosschleife werden verschiedene Schritte ausgeführt, wie z.B. *moveBall()* und *checkForCollisionsWithWall()*. Am Ende jedes Durchgangs wird die *pause()* Methode aufgerufen. Diese wartet einfach eine vorgegeben Anzahl von Millisekunden. In diesem Fall also 40 ms, was einer Framerate von 25 fps entspricht.

Gehen wir jetzt weiter die Methoden im Einzelnen durch. In der *setup()* Methode legen wir die Größe des Fensters fest, danach setzen wir die Hintergrundfarbe auf grün, und schließlich kreiiren wir einen Ball:

```
private void setup() {
    setSize(WIDTH, HEIGHT);

    setBackground(Color.GREEN);

    ball = new GOval(BALL_SIZE, BALL_SIZE);
    ball.setFilled(true);
    add(ball, WIDTH / 2, HEIGHT / 2);
}
```

Was auffallen sollte hier ist, dass es "ball =" und nicht "GOval ball =" heißt. Dazu gleich mehr. Die nächste Methode ist die *moveBall()* Methode:

```
private void moveBall() {
    ball.move(vx, vy);
}
```

Die ist ganz einfach, wir rufen einfach die *move()* Methode der Klasse GOval auf. Interessant auch hier wieder, weder *ball* noch *vx* oder *vy* werden deklariert. Und schließlich sehen wir uns die *checkForCollisionsWithWall()* Methode an:

```
private void checkForCollisionsWithWall() {
    double x = ball.getX();
    double y = ball.getY();
    if ((x < 0) || (x > WIDTH)) {
        vx = -vx;
    }
    if ((y < 0) || (y > HEIGHT)) {
        vy = -vy;
    }
}
```

Wir lassen uns die momentane *x*- und *y*-Position des Balls geben, und testen ob diese innerhalb des Spielfeldes ist. Falls nicht, dann ändern wir das Vorzeichen der Geschwindigkeit. Wenn man möchte, könnte man die Geschwindigkeit bei jeder Kollision ein wenig reduzieren, tun wir aber nicht.

Kommen wir jetzt zu dem Problem mit der fehlenden Deklaration von *ball*, *vx* und *vy*. Bisher kennen wir nur *lokale* Variablen. Das Problem mit lokalen Variablen ist, dass sie nur innerhalb einer Methode gültig sind. In unserem Billard Beispiel, brauchen wir aber den Ball in drei Methoden: der *setup()*, der *moveBall()*, und der *checkForCollisionsWithWall()* Methode. Offensichtlich können wir also für den Ball (und auch *vx* und *vy*) keine lokale Variable verwenden. Anstelle verwenden wir eine *Instanzvariable*. Instanzvariablen werden am Anfang der Klasse deklariert, vor der run() Methode, und ganz wichtig, außerhalb der run() Methode (oder irgendeiner anderen Methode):

```
public class Billiards extends GraphicsProgram {
    // instance variables
    private GOval ball;
    private int vx = 4;
    private int vy = -3;

    public void run() {
        ...
    }
    ...
}
```

Der Vorteil von Instanzvariablen ist, dass in jeder Methode auf sie zugegriffen werden kann. Im nächsten Kapitel werden wir noch mehr zu Instanzvariablen erfahren.

Events

Also Animationen waren gar nicht so schwer. Kommen wir zu den Spielen: unsere Spiele sollen mit der Maus steuerbar sein. Dazu müssen wir zwei Sachen tun, wir müssen dem Programm sagen,

1. dass es auf die Maus hören soll und
2. welche der Maus Events uns interessieren, z.B. ob die Maustaste gedrückt wurde oder die Maus bewegt wurde.

Das erste erreichen wir indem wir am Anfang unseres Programms die Methode

```
addMouseListeners();
```

aufrufen. Das zweite erreichen wir indem wir eine von zwei Methoden überschreiben: Wenn wir hören wollen, ob die Maustaste gedrückt wurde, dann verwenden wir

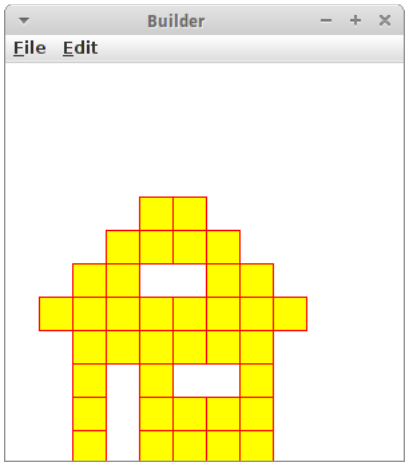
```
public void mousePressed(MouseEvent e) { ... }
```

oder wenn wir hören wollen ob sich die Maus bewegt hat, dann verwenden wir

```
public void mouseMoved(MouseEvent e) { ... }
```

Wir können natürlich auch beide verwenden.

Übung: Builder



Builder ist von Lego inspiriert: wir haben kleine Klötzchen (blocks) und die können wir beliebig auf dem Bildschirm setzen, in dem wir mit der Maus an die Position klicken wo der Klotz hin soll. Der Programmcode dafür ist total einfach. Als erstes müssen wir dem Programm sagen, dass wir auf die Maus hören möchten:

```
public void run() {
    addMouseListeners();
}
```

Danach müssen wir nur noch sagen, was denn passieren soll, wenn die Maus geklickt wurde:

```
public void mousePressed(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    GRect block = new GRect(BLOCK_SIZE, BLOCK_SIZE );
    block.setColor(Color.RED);
    add(block, x, y);
}
```

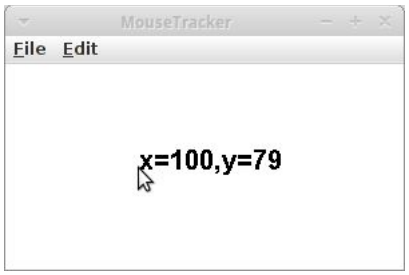
Als Erstes benötigen wir die x- und y-Position der Maus: die erhalten wir vom *MouseEvent* e. Der MouseEvent hat zwei Methoden, *getX()* und *getY()* und diese beiden Methoden geben uns die x- und y-Position der Maus. Sobald wir die haben kreieren wir ein neues GRect und platzieren es an die Position der Maus.

Mit einem kleine Trick, kann man die Position der Klötzchen "quantisieren":

```
x = x / BLOCK_SIZE * BLOCK_SIZE;
```

Das funktioniert allerdings nur wenn *x* vom Datentyp *int* ist. Warum funktioniert das?

Übung: MouseTracker



Was wir noch brauchen, ist der Mausbewegung zu verfolgen. Wie das geht demonstriert das folgende Programm. Wir beginnen wieder mit der *run()* Methode:

```
private GLabel lbl;

public void run() {
    lbl = new GLabel("");
    lbl.setFont("Arial-bold-20");
    add(lbl, 0, 0);
    addMouseListeners();
}
```

Um die Position der Maus anzuzeigen wollen wir einen GLabel namens *lbl* verwenden. Der ist eine Instanzvariable und muss initialisiert werden und zu unserem Canvas hinzugefügt werden. Danach sagen wir unserem Programm wieder, dass wir auf Maus Ereignisse hören möchten. In diesem Beispiel wollen wir wissen ob die Maus sich bewegt hat, deswegen überschreiben wir die *mouseMoved()* Methode:

```
public void mouseMoved(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    lbl.setLabel("x=" + x + ",y=" + y);
    lbl.setLocation(x, y);
}
```

Wir holen uns wieder die x- und y-Position der Maus, ändern den Text des Labels mit der *setLabel()* Methode, und verschieben den Label mit der *setLocation()* Methode an die Postion der Maus. Und das war's auch schon.

RandomGenerator

Für viele Spiele benötigen wir Zufallszahlen. Dafür gibt es die Klasse *RandomGenerator*. Die kann nicht nur Zufallszahlen, sondern auch Zufallsfarben erzeugen. Der Code

```
RandomGenerator rgen = new RandomGenerator();
...
double width = rgen.nextDouble(0, 150);
Color col = rgen.nextColor();
```

zeigt wie man Zufallszahlen und Zufallsfarben erzeugt. Es gibt auch die Methoden nextInt(a,b) und nextBoolean().

GCanvas

An dieser Stelle macht es noch Sinn eine Klasse vorzustellen, die wir noch nicht explizit kennengelernt haben: die Klasse GCanvas. Verwendet haben wir sie aber schon, und zwar als wir unser GRect *lisa* hinzugefügt haben mit

```
add(lisa, 70, 50);
```

Die Frage die wir uns damals nämlich hätten stellen sollen ist wozu? Und die Antwort ist natürlich: zum GCanvas. GCanvas ist unser Filzbrett. Wenn wir was hinzufügen können, stellt sich als nächstes gleich die Frage, können wir auch was wegnehmen? Und was können wir denn sonst noch so machen mit dem GCanvas? Auch dafür gibt es eine Liste von Methoden:

- **add(object):** fügt eine GObject zum GCanvas hinzu
- **add(object, x, y):** fügt eine GObject zum GCanvas an der Position x, y hinzu
- **remove(object):** entfernt das GObject object von GCanvas
- **removeAll():** entfernt alle GObjects von GCanvas
- **getElementAt(x, y):** gibt uns das vorderste GObject an der Position x, y, falls dort eines ist
- **setBackground(c):** ändert die Hintergrundfarbe des GCanvas

Außerdem, ist ein GCanvas auch ein GObject, d.h. alles was GObject kann, kann auch GCanvas.

Review

Eigentlich haben wir dieses Kapitel gar nicht so viel gemacht. Aber das ist nicht ganz richtig, in den Projekten werden wir gleich sehen, dass wir schon wirklich coole Sachen programmieren können. Was haben wir bisher in diesem Kapitel gelernt? Wir wissen jetzt was

- Methoden
- Paremeter
- Rückgabewert
- und lokale Variablen

sind. Ausserdem haben wir

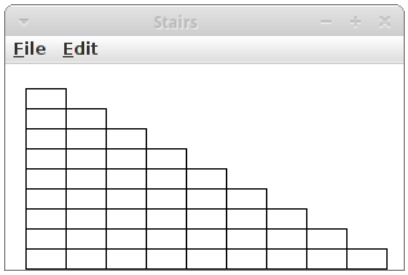
- GObject
- GCanvas
- und den RandomGenerator

kennengelernt. Das wichtigste in diesem Kapitel war, dass wir Animation mit dem *GameLoop* erreichen können und auf Maus Events reagieren können.

Projekte

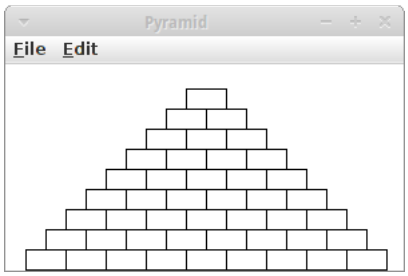
Die Projekte in diesem Kapitel fangen an richtig Spaß zu machen. Los geht's.

Stairs



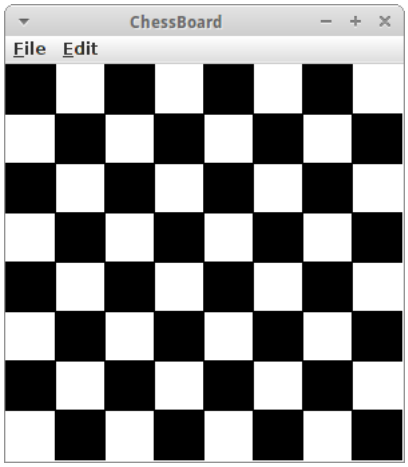
Unser erstes Projekt ist eine kleine Treppe. Das Problem ist ganz ähnlich wie das Wall Problem. Daher macht es auch hier Sinn nach dem Top-Down Ansatz anzunehmen es gibt eine Methode drawRowOfNBricks*(int n)* gibt. Wir sollten auch darauf achten keine *magic numbers* zu verwenden, sondern nur Konstanten.

Pyramid



Die Pyramide ist fast identisch zur Treppe. Der einzige Unterschied ist, dass die Stufen immer einen halben Stein versetzt sind. Die Treppe soll neun Steine in der untersten Reihe haben. Eigentlich brauchen wir den Code vom letzten Beispiel nur geringfügig zu ändern.

ChessBoard



Kehren wir zu unserem Schachbrett zurück. Dieses Mal wollen wir aber den Top-Down Ansatz verwenden. Dazu gibt es mehrere Ansätz, aber einer wäre eine Methode namens *drawOneRow()* zu deklarieren. Hier muß man sich genau überlegen welche Parameter man an die Methode übergibt. Man könnte auch zwei Methoden haben, eine für gerade und eine für ungerade Zeilen.

RGBColor

Wir haben ja schon den Regenbogen gezeichnet, allerdings noch sehr "händisch". Jetzt wollen wir die HSV Farbpalette zeichnen [5]. In Java kann man eine beliebige Farbe mittels

```
Color col = new Color(r, g, b);
```

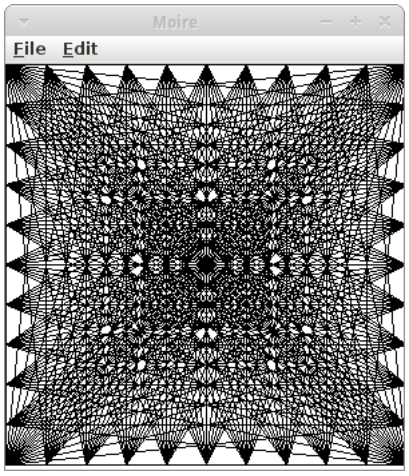
erzeugen, wobei die Variablen r, g, und b jeweils für den rot, grün und blau Anteil stehen. Diese müssen Werte zwischen 0 und 255 sein. Also z.B. Rot wird durch

```
Color colRed = new Color(255, 0, 0);
```

erzeugt.

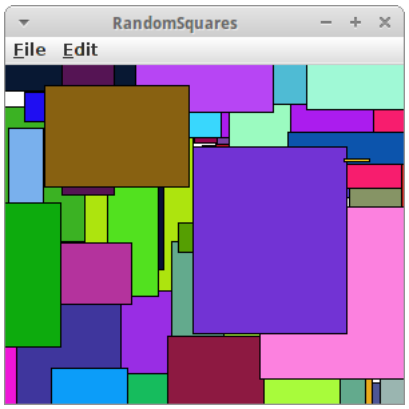
Wenn wir uns die Farben in der HSV Farbpalette genau ansehen bemerken wir, dass es beginnend mit Rot über die Farben Gelb, Grün, Cyan, Blau, Magenta wieder zu Rot zurück kommt. Insgesamt gibt es also 6 Farbabstufungen. Den ersten Übergang von Rot nach Gelb, könnte man z.B. durch folgende Zeilen erreichen:

```
for (int x = 0; x < 255; x++) {
    Color col = new Color(255, x, 0);
    GLine line = new GLine(x, 0, x, HEIGHT);
    line.setColor(col);
    add(line);
}
```



Moire

Der Moiré-Effekt [6] ist normalerweise eher unerwünscht, aber er kann auch ganz hübsch sein. Man teilt zunächst die Länge und Breite in gleiche Teile auf, z.B. 11. Dann zeichnet man von jedem Punkt oben zu jedem Punkt unten eine Linie und das Gleiche von links nach rechts. Am besten man zeichnet sich erst einmal auf einem Stück Papier auf wie man selbst das zeichnen würde. Es läuft auf zwei verschachtelte *for* Scheifen hinaus.

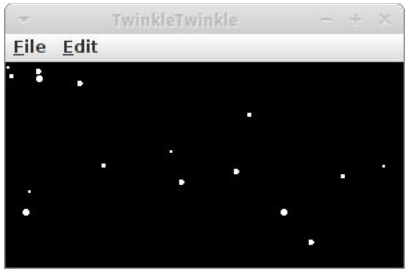


RandomSquares

Wir wollen uns weiter künstlerisch betätigen. Es geht darum verschieden farbige Rechtecke von zufälliger Größe und Position zu zeichnen. Man benutzt natürlich den RandomGenerator. Zunächst lässt man sich eine zufällige Breite und Höhe für ein GRect geben. Diese sollten vielleicht nicht zu groß oder zu klein sein. Dann gibt man dem Rechteck eine zufällige Farbe mit

```
rect.setFillColor(rgen.nextColor());
```

Un schließlich plaziert man es noch an eine zufällige x und y Position. Das Ganze kommt dann in eine Endlosschleife und man wartet vielleicht 100 ms.



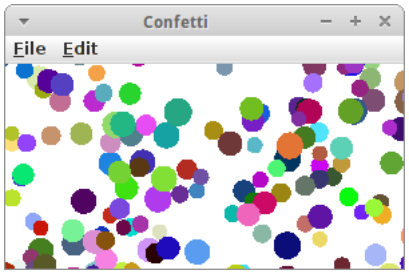
TwinkleTwinkle

Bei TwinkleTwinkle geht es darum einen zufälligen Sternenhimmel zu generieren. Die Sterne sind GOvals mit eine zufälligen Größe zwischen 1 und 4 Pixeln. Die werden zufällig auf dem Canvas verteilt. Vorher sollte man noch den Hintergrund mittels

```
setBackground(Color.BLACK);
```

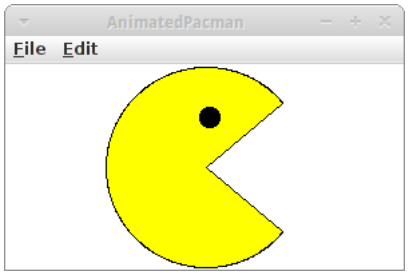
auf Schwarz setzen. Das Ganze kommt dann in eine Endlosschleife und man wartet vielleicht 500 ms bis zur Erzeugung des nächsten Sterns.

Confetti



Wir alle lieben Confetti. Dabei sind die ganz einfach zu machen, entweder mit einem Locher oder mit GOvals. Die Confetti können alle gleich groß sein (z.B. 20 Pixel), müssen es aber nicht. Aber auf jeden Fall haben sie verschiedene Farben, wieder ein Fall für den RandomGenerator. Und natürlich sollte die Postion der Confetti zufällig sein, also das Ganze kommt wieder in eine Endlosschleife.

AnimatedPacman

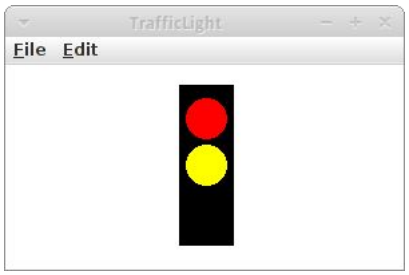


Wir haben ja schon vor zwei Kapiteln unseren ersten PacMan gemalt. Der war aber noch recht statisch. Wir wollen PacMan jetzt animieren. Dazu ist es nützlich zu wissen, dass es die beiden Kommandos für GArcs

```
pacman.setStartAngle(angle);
pacman.setSweepAngle(360 - 2 * angle);
```

gibt. Wenn wir jetzt die *angle* Variable zwischen 0 und 40 variieren lassen, und das alle 50 ms tun, dann erscheint das so wie wenn PacMan animiert wäre.

TrafficLight



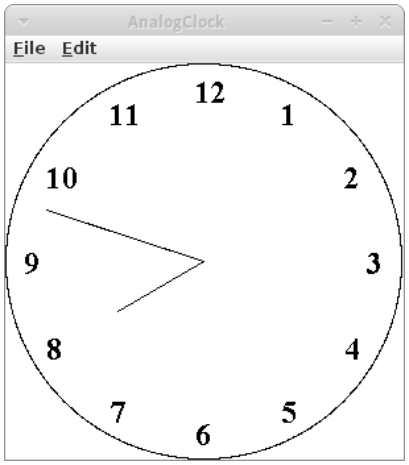
Auch die Ampel haben wir bereits vor zwei Kapiteln gezeichnet. Jetzt wollen wir aber die Ampel animieren. Die Ampel beginnt mit Rot, geht dann über zu Rot-Gelb, gefolgt von Grün. Schließlich geht es wieder über Gelb zurück nach rot. Der Übergang sollte jeweils eine Sekunde dauern, und das Ganze sollte wieder in einer Endlosschleife verpackt sein. Man könnte z.B. Instanzvariablen für die Lichter haben, und diese dann mittels

```
if (currentLight == 0) {
    redLight.setColor(Color.RED);
    yellowLight.setColor(Color.BLACK);
    greenLight.setColor(Color.BLACK);
} ...
```

ein- und ausschalten. Jetzt muss man sich noch überlegen wie man zwischen den verschiedenen Zuständen hin- und herschaltet. Das kann man sehr geschickt mit dem Remainder Operator % machen:

```
    currentLight++;
    currentLight = currentLight % 4;
```

AnalogClock



In der *setup()* Methode zeichnen wir den Kreis und die Uhrzeiten mittels GLabels. Dabei kann man die GLabels von Hand setzen, oder mittels Sinus und Kosinus ausrechnen wo sie denn am besten hinpassen. Beides dauert ungefähr gleich lange, das zweitere bedarf aber etwas mehr Gehirnschmalz.

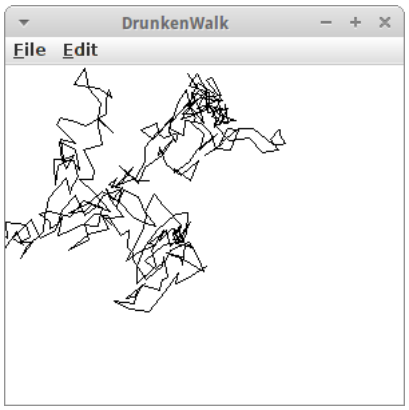
Allerdings wenn es darum geht die Zeiger zu zeichnen, kommen wir am Sinus und Kosinus [7] nicht mehr vorbei.

```
    private void drawSeconds(int seconds) {
        double radians = 2 * Math.PI * seconds / 60;
        double lengthSecondHand = 250;
        double x = SIZE / 2 + Math.sin(radians) *
lengthSecondHand / 2;
        double y = SIZE / 2 - Math.cos(radians) * lengthSecondHand / 2;
        secondsHand.setEndPoint(x, y);
    }
```

dabei handelt es sich bei *secondsHand* um eine GLine

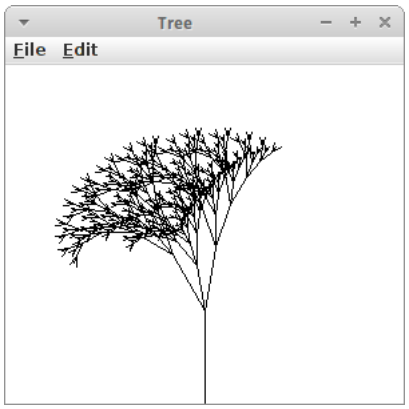
```
private GLine secondsHand;
```

die als Instanzvariable deklariert wurde. Wie wir an die Stunden, Minuten und Sekunden kommen, haben wir ja bereits in dem Projekt "Time" des letzten Kapitel gesehen. Da die Uhr natürlich animiert sein soll, brauchen wir natürlich eine Endlosschleife, mit einer Pause von einer Sekunde, evtl sogar nur einer halben Sekunde.



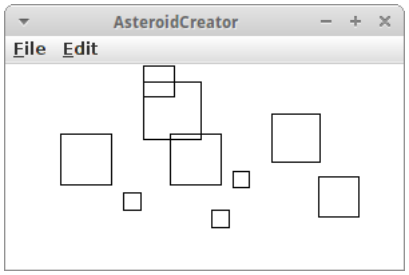
DrunkenWalk

Wenn Karel mal einen über den Durst getrunken hat, dann läuft er nicht mehr so gerade. In diesem Beispiel fängt er in der Mitte an. Einmal pro Sekunde macht er dann einen Schritt, zufälliger Distanz in eine beliebige Richtung. Wobei er natürlich in einem Schritt nicht beliebig weit kommt. Mit GLines zeichnen wir die Schlangenlinien nach, um Karel am nächsten Morgen zu zeigen, dass er doch besser ein Taxi hätte nehmen sollen.



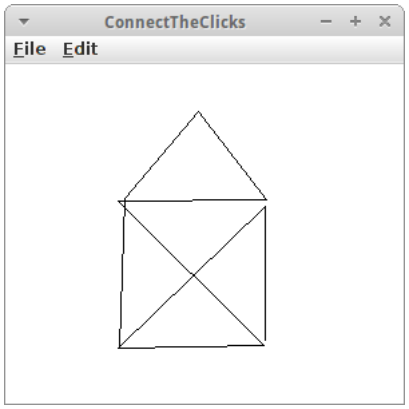
Tree*

Bäume zu zeichnen stellt sich als relativ schwierig heraus. Eine beliebt Technik um dieses Problem zu lösen ist die Rekursion. Da wir aber noch nichts von Rekursion gehört haben, versuchen wir einen Baum ohne Rekursion zu zeichnen. Nach dieser Erfahrung, sind wir vielleicht motivierter im nächsten Semester die Geheimnisse der Rekursion kennen zu lernen.



AsteroidCreator

Das Arcade-Spiel 'AsteroidCreator' war ein absoluter Hit in den späten 80er Jahren. In dem Spiel geht es darum, an der Stelle an der die Benutzerin mit der Maus klickt einen Asteroiden zu zeichnen. Dabei sind Asteroiden einfach GRects unterschiedlicher, zufälliger Größe mit schwarzem Rand. Wir müssen also wieder die *addMouseListeners()* Methode im Setup aufrufen. In der *mousePressed()* Methode malen wir dann einfach ein Rechteck an der Position an der die Benutzerin mit der Maus geklickt hat.



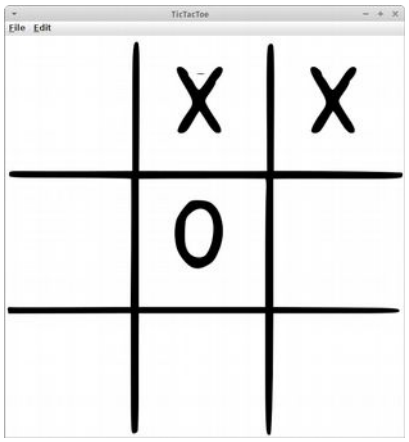
ConnectTheClicks

Ganz ähnlich wie das vorherige Spiel war auch 'ConnectTheClicks®' sehr populär in den späten 70er Jahren. Dabei handelt es sich um ein Spiel in dem der Benutzer mit der Maus an eine Stelle klickt, und diese dann mit der vorhergehenden verbunden wird. Was das Spiel etwas komplizierter macht, ist dass man sich merken muss, an welche Stelle der Benutzer vorher geklickt hat. Dazu verwenden wir einfach zwei Instanzvariablen:

```
private int x0 = -1;
private int y0 = -1;
```

Wenn wir die Variablen mit dem Wert "-1" initialisieren, dann ist das ein kleiner Trick mit dem wir feststellen können, ob das der erste Klick ist. Denn dann dürfen wir noch keine Linie ziehen! Ansonsten, zeichnen wir einfach eine Linie (GLine) bei jedem Klick von der alten Position auf die neue Maus-Position.

TicTacToe



TicTacToe kennt jeder aus dem Kindergarten oder der Schule, notfalls kann man in der Wikipedia dazu folgendes nachlesen [9]:

"Auf einem quadratischen, 3×3 Felder großen Spielfeld setzen die beiden Spieler abwechselnd ihr Zeichen (ein Spieler Kreuze, der andere Kreise) in ein freies Feld. Der Spieler, der als Erster drei Zeichen in eine Zeile, Spalte oder Diagonale setzen kann, gewinnt."

Im *setup()* zeichnen wir den Hintergrund, am einfachsten ist das als Bild, und sorgen mittels *addMouseListeners()*, dass wir auf Maus Ereignisse hören. Dazu müssen wir noch die *mousePressed()* Methode hinzufügen. Dort müssen wir dann einfach abwechselnd ein "X" oder ein "O" malen, je nachdem wer dran ist. Wie weiß man wer dran ist? Das kann man z.B. über eine Instanzvariable machen:

```
private int currentPlayer = 1;
```

Die darf zwei Werte haben, '1" für Spieler eins und "2" für Spieler zwei. Das Umschalten zwischen den beiden Spielern geht dann ganz einfach über:

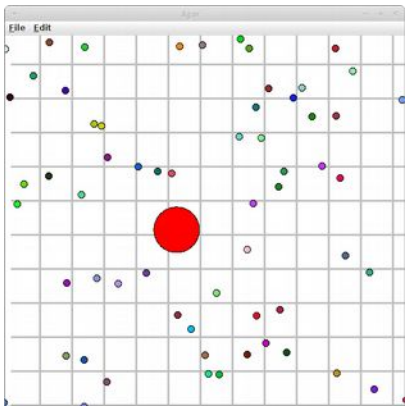
```
if (currentPlayer == 1) {
    ...
    currentPlayer = 2;
} else {
    ...
    currentPlayer = 1;

}
```

Eine kleine Sache noch die ganz praktisch ist: man kann natürlich einfach die "X" und "O" da malen wo der Nutzer geklickt hat. Das schaut dann aber etwas schepps aus. Hier kann man einmal von der Integer Division profitieren:

```
public void mousePressed(MouseEvent e) {
    int x = e.getX();
    int i = x / CELL_WIDTH;
    int xPos = i * CELL_WIDTH;
    …
}
```

Challenges

Agrar



Das Spiel Agrar ist inspiriert von dem Spiel Agar.io, Dabei geht es laut Wikipedia [10] darum:

"... eine Zelle zu navigieren, die dadurch wächst, dass sie Pellets und andere Zellen frisst. Sie darf dabei jedoch nicht selbst von größeren Zellen gefressen werden."

Unsere Version des Spiels ist etwas einfacher, es gibt nämlich nur eine Zelle, und die kann nur Pellets futtern. Aber das ist auch schon recht anspruchsvoll.

Wir brauchen zunächst eine Instanzvariable für die Zelle:

```
private GOval cell;
```

Die initialisieren wir in der *setup()* Methode, dort fügen wir auch den MouseListener hinzu. Als nächstes benötigen wir den *GameLoop*:

```
while (true) { // game loop
    createRandomFood();
    checkForCollision();
    pause(DELAY);
}
```

in dem wir an einer zufälligen Position ein Pellet, also ein GOval mit zufälliger Farbe erzeugen. Als nächstes schauen wir ob es zwischen Zelle und Pellet zu einer Kollision gekommen ist. Wie wissen wir ob es zu einer Kollision gekommen ist? Dafür gibt es die Methode *getElementAt()*:

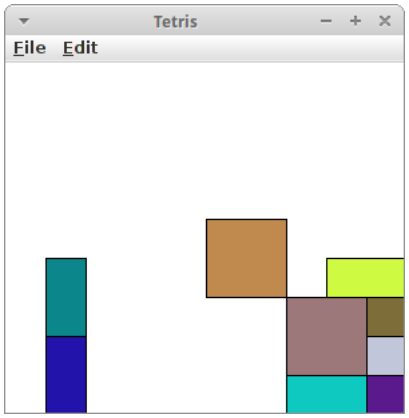
```
GObject collisonObject = getElementAt(x, y);
if ( (collisonObject != null) && (collisonObject != cell) ) {
    double w = cell.getWidth();
    cell.setSize(w + 1, w + 1);
    remove(collisonObject);
}
```

Diese Methode schaut ob sich an der Position x,y etwas befindet. Wenn da nichts ist, dann gibt die Methode den Wert "null" zurück. Ansonsten gibt sie uns das Objekt zurück, das sich an dieser Position befindet. Das könnte ein Pellet sein, das könnte aber auch die Zelle selbst sein. Deswegen müssen wir checken, dass es nicht "null" ist, und dass es nicht die Zelle ist. Da es ja sonst nichts gibt, wissen wir jetzt, dass das *collisonObject* ein Pellet sein muss. Das Pellet "essen" wir dann, was bedeutet, dass die Zelle fetter wird, und das die Pellet verschwindet.

Und natürlich müssen wir noch die *mouseMoved()* Methode implementieren. Die ist aber ganz einfach, weil wir einfach die Zelle an die Position der Maus bewegen:

```
public void mouseMoved(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    cell.setLocation(x, y);
}
```

Tetris



Tetris ist inzwischen ein Spiele Klassiker. Ursprünglich wurde es vom russischen Programmierer Alexei Paschitnow programmiert [11]. Tetris besitzt verschiedene Steinformen die etwas lateinischen Buchstaben ähneln (I, O, T, Z und L). Die Spieler müssen die einzelnen Steine die vom oberen Rand des Spielfelds herunterfallen in 90-Grad-Schritten drehen und sie so verschieben, dass sie am unteren Rand horizontale, möglichst lückenlose Reihen bilden. Wie üblich beschränken wir uns auf eine etwas einfachere Version, in der es nur vier Steinformen gibt: einen Einer-Block, einen horizontalen und einen vertikalen Zweier-Block, und einen Vierer-Block. Drehen kann man die Blöcke in unserer einfachen Version nicht.

In der *setup()* Methode legen wir einen neuen Stein an:

```
private void setup() {
    createNewBrick();
}
```

Die Methode *createNewBrick()* legt einen zufälligen neuen Stein an. Abhängig von einer Zufallszahl wird jeweils eine andere Steinform erzeugt:

```
private void createNewBrick() {
    switch (rgen.nextInt(0, 3)) {
    case 0:
        brick = new GRect(WIDTH / 2, 0, BRICK_SIZE, BRICK_SIZE * 2);
        break;
    case 1:
        ...
    }
    brick.setFilled(true);
    brick.setFillColor(rgen.nextColor());
    add(brick);
}
```

Der Stein *brick* muss natürlich eine Instanzvariable sein, sonst funktioniert das nicht. Das Gleiche gilt für *rgen*. Damit die Steine dann anfangen herunterzufallen, benötigen wir einen *GameLoop*:

```
while (true) {
    moveBrick();
    checkForCollision();
    pause(DELAY);
}
```

Die Methode *moveBrick()* bewegt den Stein einfach um eine Steinbreite nach unten. Die Methode *checkForCollision()* stellt fest ob der Stein weiter fallen darf. Ist der Stein unten angekommen, dann darf er nicht weiter fallen. Der Trick das zu bewerkstelligen ist eigentlich ganz einfach: wir kreieren einen neuen Stein, und lassen den alten da wo er ist:

```
private void checkForCollision() {
    if (brick.getY() > getHeight() - brick.getHeight()) {
        createNewBrick();
    } ...
}
```

Der andere Fall der eintreten kann, ist wenn unterhalb des fallenden Steins ein anderer Stein ist, dann darf er auch nicht weiter fallen. Da müssen wir mit der Methode *getElementAt()* arbeiten:

```
int x = brick.getX() + 1;
int y = brick.getY() + BRICK_SIZE;
GObject obj = getElementAt(x, y);
if (obj != null) { ... }
```

Diese Methode sagt uns, ob es an der Position x,y ein GObject gibt. Falls nein, dann gibt die Methode "null" zurück. Auch hier kreieren wir einfach einen neuen Stein.

So, jetzt fallen also unsere Stein munter runter. Es fehlt nur noch die Steuerung durch die Tastatur. Die Tastatur hat Tasten, auf Englisch "Keys" und wenn eine Taste gedrückt wird dann kommt es zu einen *KeyEvent*. Das ist vollkommen analog zu den Maus Events. Deswegen müssen wir also in der setup() Methode noch einen KeyListener hinzufügen:

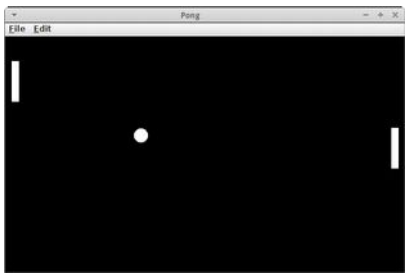
```
addKeyListeners();
```

und die Methode *keyPressed()* hinzufügen:

```
public void keyPressed(KeyEvent e) {
    int cc = e.getKeyCode();
    switch (cc) {
    case 37:
        // move left
        brick.move(-BRICK_SIZE, 0);
        break;
    case 39:
        // move right
        brick.move(BRICK_SIZE, 0);
        break;
    }
}
```

Über den Keycode erfahren wir welche Taste gedrückt wurde: für die linke Pfeiltaste ist der Keycode die 37, für die rechte die 39. Und damit ist unsere einfache Tetris Version schon fertig.

Pong



Pong wurde 1972 von Atari veröffentlicht und gilt als Urvater der Videospiele [8]. Es ist ein Spiel für zwei Spieler, die versuchen ähnlich wie beim Tischtennis (ping-pong) einen Ball hin und her zu spielen. Zunächst benötigen wir drei Instanzvariablen

```
private GOval ball;
private GRect leftPaddle;
private GRect rightPaddle;
```

für den Ball und die beiden Paddles. Dann gibt es natürlich wieder einen *GameLoop*:

```
while (true) {
    moveBall();
    checkForCollision();
    pause(DELAY);
}
```

Die Methode *moveBall()* bewegt den Ball einfach um einen gewissen Betrag in x und y Richtung.

Die Methode *checkForCollision()* tut zwei Dinge: sie stellt fest ob es eine Kollision mit der Wand gab oder ob es eine Kollision mit einem der Paddles gab. Falls der Ball das Spielfeld nach oben oder unten verlassen möchte, wird er einfach reflektiert, falls er aber nach links oder rechts verschwindet, ist die Runde beendet. Falls der Ball mit den Paddles kollidiert wird er auch einfach reflektiert.

Für die Kollisionen mit den Paddles verwenden wir wieder die *getElementAt()* Methode. Das Reflektieren ist eigentlich auch ganz einfach. Dazu müssen wir aber noch eine Instanzvariable für die Geschwindigkeit einführen:

```
private int vx = 2;
private int vy = 3;
```

Reflektion bedeutet dann einfach:

```
vy = -vy;
```

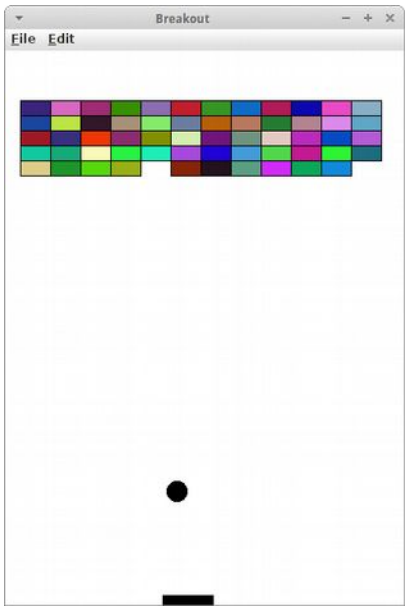
falls der Ball oben oder unten reflektiert werden soll. Natürlich muss die *moveBall()* Methode diese Variablen verwenden.

Die Kontrolle der Paddles soll über die Tastatur erfolgen, also müssen wir wieder einen KeyListener hinzufügen und die Methode *keyPressed()* hinzufügen:

```
public void keyPressed(KeyEvent e) {
    char c = e.getKeyChar();
    switch (c) {
    case 'p': // right paddle up
        rightPaddle.move(0, -PADDLE_VY);
        break;
    case 'l': // right paddle down
        ...
}
```

Wir könnten wieder mit *getKeyCode()* arbeiten, aber *getKeyChar()* ist hier viel praktischer. Der erste Spieler kontrolliert seinen Paddle über die Tasten 'q' und 'a', der zweite Spieler über die Tasten 'p' und 'l'.

Breakout



Das erste Breakout-Spiel wurde im April 1976 von Atari vorgestellt. Das Spielprinzip stammt von Nolan Bushnell. Steve Jobs, der damals bei Atari arbeitete, überredete seinen Freund Steve Wozniak (damals bei HP), dieses Spiel zu konstruieren. Steve Jobs bekam für das Spiel Breakout von Nolan Bushnell 5.000 Dollar bezahlt. Er gab seinem Freund Steve Wozniak, die Hälfte, also 350 Dollar [12].

Das Spielfeld besteht aus Mauersteinen, einem Ball und einem Schläger. Der Ball bewegt sich durch das Spielfeld und wenn er auf einen Mauerstein trifft, dann verschwindet dieser. Von den Wänden und dem Schläger wird der Ball reflektiert. Außer bei der unteren Wand, also wenn der Ball nicht mit dem Schläger getroffen wird, dann ist das Spiel zu Ende. In unserem Spiel soll der Schläger durch die Maus kontrolliert werden.

Als Instanzvariablen benötigen wir wieder den Ball und den Schläger:

```
private GOval ball;
private GRect paddle;
```

Dann folgt die *setup()* Methode. Dort wollen wir den Ball und den Schläger initialisieren und auch die Mauer malen.

```
private void setup() {
    createBall();
    createPaddle();
    createBricks();
}
```

Beim Malen der Mauer, können wir aber die Methode *drawRowOfNBricks(int n)* aus dem Projekt Stairs borgen, damit wir es ganz einfach. Evtl. wollen wir die Bricks noch verschieden farbig machen, dann sieht das Spiel gleich viel besser aus.

Nach dem Setup kommen wir wieder zum *GameLoop*:

```
while (true) {
    moveBall();
    checkForCollision();
    pause(DELAY);
}
```

Die *moveBall()* Methode ist identisch zu der in Pong. Auch *checkForCollision()* ist sehr ähnlich

```
private void checkForCollision() {
    checkForCollisionWithWall();
    checkForCollisionWithPaddleOrBricks();
}
```

Was ein bisschen mehr Arbeit benötigt ist die *checkForCollisionWithPaddleOrBricks()* Methode. Wir verwenden wieder die *getElementAt()* Methode, schauen nach ob das Objekt nicht null ist. Dann kann es sich nur um den Schläger oder einen Mauerstein handeln, also

```
GObject obj = getElementAt(x, y);
if (obj != null) {
    if (obj == paddle) {
        vy = -vy;
    } else {
        // must be brick:
        remove(obj);
        vy = -vy;
    }
}
```

Damit wäre der GameLoop erledigt, bleibt noch die Bewegung des Schlägers durch die Maus. Dazu müssen wir im Setup die *addMouseListeners()* Methode aufrufen und die Methode *mouseMoved()* implementieren:

```
public void mouseMoved(MouseEvent e) {
    int x = e.getX();
    paddle.setLocation(x, getHeight() - PADDLE_SEPARATION);
}
```

Und das war's auch schon.

Fragen

1. Es gibt Methoden mit Rückgabewert und welche ohne. Woran erkennt man, dass eine Methode keinen Rückgabewert hat?

2. Manche Methoden haben Parameter (auch Übergabeparameter genannt). Was sind Parameter?

3. Was haben Parameter und Kilometer gemeinsam?

4. Wir haben mehr als einmal den RandomGenerator 'rgen' verwendet. Wie heißt das Kommando um eine Zufallszahl (int) zwischen 1 und 6 zu erzeugen?

5. Wenn man einen primitiven Datentyp an eine Methode als Parameter übergibt, wird dieser im Original oder als Kopie übergeben?

6. Nennen Sie vier Unterklassen (Kinderklassen) der Klasse GObject.

7. Zeichnen Sie Diagramm das grob die Klassenhierarchie der GObjects darstellt.

8. Wir haben den Top-Down Ansatz kennengelernt. Dieser gibt Regeln bzgl der Namen von Methoden, wie viele Zeilen Code eine Methoden haben sollte, etc. Nennen Sie zwei dieser Richtlinien.

9. Analysieren Sie das Spiel 'Agar.io' mit Hilfe des Top-Down (Von Oben Nach Unten) Ansatzes. Es geht nur um die Grobstruktur (high-level), nicht um detaillierten Code.

10. Die Methode addThirteen() im folgenden Code funktioniert nicht wie erwartet. Was ist das Problem? Wie würden Sie es lösen?

```
private void addThirteen(int x ) {
    x += 12;
}
public void run() {
    int x = 4;
    addThirteen( x );
    println( "x = " + x );
}
```

11. Das Arcade-Spiel 'RandomCircles' war ein absoluter Hit in den späten 80er Jahren. Es handelt sich dabei um ein Spiel, das an der Stelle an der die Benutzerin mit der Maus klickt einen bunten Kreis zeichnet. Es wäre etwas zu viel verlangt die Vollversion des Spiels zu implementieren, aber es ist relativ einfach, den Code zu schreiben, der
    1) einen farbigen Kreis auf dem Bildschirm zeichnet und
    2) den Kreis dort zeichnet, wo die Benutzerin mit der Maus geklickt hat.

    In einem ersten Schritt erstellen Sie eine Klasse namens GCircle, die GOval erweitert (extends) und ein Kreis mit zufälliger Größe und Farbe zeichnet.

    In einem zweiten Schritt, schreiben Sie eine GraphicsProgram, das auf Mausklicks hört und einen Kreis an die Stelle zeichnet, an die die Benutzerin mit der Maus geklickt hat.

    Evtl ist die folgende Information hilfreich:

    ```
    private RandomGenerator rgen = RandomGenerator.getInstance();
    ```

    Der RandomGenerator hat u.a. folgende Methoden:

    ```
    int nextInt( int low, int high )
      Color nextColor()
    ```

Referenzen

Die Referenzen relevant für dieses Kapitel sind die gleichen wie in Kapitel 2. Weitere Details zur ACM Grafik Bibliothek kann man auf den Seiten der ACM Java Task Force [1] finden. Viele weitere Beispiele findet man im Tutorial [2], dem Buch von Eric Roberts [3] und der Stanford Vorlesung 'Programming Methodologies' [4] oder sind von ihnen inspiriert.

[1] ACM Java Task Force, cs.stanford.edu/people/eroberts/jtf/

[2] ACM Java Task ForceTutorial , cs.stanford.edu/people/eroberts/jtf/tutorial/index.html

[3] The Art and Science of Java, von Eric Roberts, Addison-Wesley, 2008

[4] CS106A - Programming Methodology - Stanford University, https://see.stanford.edu/Course/CS106A

[5] HSL and HSV, https://en.wikipedia.org/w/index.php?title=HSL_and_HSV&oldid=694879918 (last visited Mar. 3, 2016).

[6] Moiré-Effekt, https://de.wikipedia.org/wiki/Moiré-Effekt

[7] Sinus und Kosinus, https://de.wikipedia.org/wiki/Sinus_und_Kosinus

[8] Pong, https://de.wikipedia.org/wiki/Pong

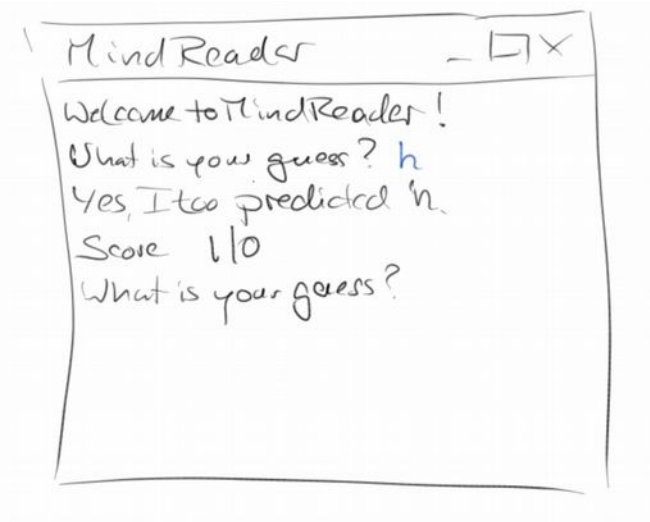
[9] Tic-Tac-Toe, https://de.wikipedia.org/wiki/Tic-Tac-Toe

[10] Agar.io, https://de.wikipedia.org/wiki/Agar.io

[11] Tetris, https://de.wikipedia.org/wiki/Tetris

[12] Breakout, https://de.wikipedia.org/wiki/Breakout_(Computerspiel)

MindReader



Dieses Kapitel beschäftigen wir uns mit zwei Themen. Wir beginnen mit Buchstaben, Zeichen und Zeichenketten, auch Strings genannt. Wir werden also wieder Konsolenprogramme schreiben, aber dieses Mal geht es vor allem um Text und Textverarbeitung.

Außerdem werden wir aber unser Verständnis von Klassen vertiefen. Wir werden sehen wie Klassen aufgebaut sind, und wir werden unsere ersten eigenen Klassen schreiben.

char

Wir beginnen mit dem 'char' Datentyp, den wir ganz kurz im vorletzten Kapitel gesehen haben. Eine Variable *c* vom Datentyp *char* legt man wie folgt an:

```
char c = 'A';
```

In einer *char* Variablen kann immer nur ein Zeichen sein. Zeichen können Buchstaben, Ziffern und alle möglichen Sonderzeichen sein. Die Apostrophen (single quote) sind u.a. deswegen notwendig, damit man Ziffern von Zahlen unterscheiden kann:

```
int x = 5;
char d = '5';
```

Im ersten Beispiel handelt es sich um die Zahl 5, im zweiten um die Ziffer '5'. Man kann auch Sonderzeichen darstellen, z.B., '.', '$', aber auch unsichtbare Zeichen wie '\n' für eine neue Zeile.

Character

Die *Character* Klasse ist eine Hilfsklasse die man ab und zu gut gebrauchen kann. Z.B. hat sie eine Methode die feststellt ob ein Buchstabe ein Großbuchstabe ist:

```
char c = 'A';
if ( Character.isUpperCase( c ) ) {
    println("Is upper case character.");
}
```

Zusätzlich gibt es noch die folgenden Methoden, die testen ob es sich bei dem Zeichen

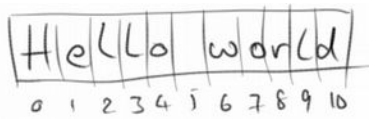
- **isDigit(c):** um eine Ziffer,
- **isLetter(c):** um einen Buchstaben,
- **isLetterOrDigit(c):** um einen Buchstaben oder eine Ziffer,
- **isWhitespace(c):** um ein Leerzeichen, Tab ('\t') oder Neu Zeile ('\n'),
- **isLowerCase(c):** um einen Kleinbuchstaben
- **isUpperCase(c):** oder um einen Großbuchstaben

handelt.

Ansonsten wird die Character Klasse aber eher selten benötigt.

String

Viel, viel wichtiger hingegen ist die *String* Klasse. Strings bezeichnet man auch als Zeichenketten, also mehrere Zeichen. Fangen wir mit ein paar Beispielen an:




```
String s1 = "Hello";
String s2 = "world";
```

Hier deklarieren wir zwei Strings, s1 und s2, und initialisieren sie auf die Werte "Hello" und "world". Wichtig sind dabei die Anführungszeichen (double quotes) um Strings von chars zu unterscheiden. Denn ein char ist immer genau ein Zeichen, Strings hingegen können ein oder mehrere Zeichen enthalten, sie können sogar leer sein:

```
String s3 = "!";
String s4 = " ";
String s5 = "";
```

Hier enthält s3 ein Zeichen, das Ausrufezeichen. Auch s4 enthält ein Zeichen, das Leerzeichen. Und s5 enthält gar kein Zeichen, s5 ist aber trotzdem ein String.

Was Strings wirklich interessant macht ist, dass wir aus alten neue machen können indem wir sie aneinanderfügen


```
String s6 = s1 + s4 + s2 + s3;
```

und auch augeben können

```
println( s6 );
```

Und ähnlich wie wir Zahlen (int und double) von der Konsole einlesen können, können wir auch Strings von Konsole einlesen:

```
String name = readLine("Enter your name: ");
```

So wie die Character Klasse, hat auch die String Klasse zahlreiche Methoden:

- **length():** gibt die Anzahl der Zeichen in einem String zurück
- **charAt(int pos):** gibt das Zeichen (char) an der Position *pos* zurück
- **equals(String s2):** stellt fest ob dieser String und der String s2 gleich sind
- **substring(int p1):** gibt den Teilstring ab der Position p1 bis zum Ende des Strings zurück
- **substring(int p1, int p2):** gibt den Teilstring von Position p1 bis zu Position p2 zurück, allerdings ohne das Zeichen an Position p2
- **indexOf(char c):** stellt fest ob das Zeichen *c* im String enthalten ist, falls ja gibt es die Position des Zeichens anderfalls -1 zurück
- **indexOf(String s2):** macht das Gleiche, sucht aber nach einem ganzen String
- **compareTo(String s2):** vergleicht zwei Strings alphabetisch, d.h. ein String der mit 'A' anfängt kommt vor einem String der mit 'B' anfängt
- **toUpperCase():** gibt die Großbuchstabenversion eines Strings zurück, natürlich gibt es auch eine toLowerCase().

Wie wir gleich sehen werden, kann man damit einiges anstellen.

Übung: Die Zeichen eines Strings ausgeben

Als erstes Beispiel iterieren wir durch die Zeichen eines Strings und geben diese auf der Konsole aus:

```
String s1 = "Hello";
for ( int i=0; i<s1.length(); i++) {
    char c = s1.charAt( i );
    println( c );
}
```

Übung: Einen String umkehren

Man kann was wir gerade gelernt haben dazu verwenden einen String umzudrehen:

```
String s1 = "stressed";
String s2 = "";
for ( int i=0; i<s1.length(); i++) {
    char c = s1.charAt( i );
    s2 = c + s2;
}
println( s2 );
```

dann wird aus dem Wort "stressed" das Wort "desserts". Worte die gleich bleiben wenn man sie umkehrt, wie z.B. "racecar" oder "rentner", nennt man auch Palindrome.

Übung: Zwei Strings vergleichen

Als nächstes wollen wir feststellen, ob zwei Strings gleich sind. Bisher haben wir dafür "==" verwendet. Das funktioniert sehr gut für int, double und auch char, aber nicht für Strings, wie das folgende Beispiel zeigt:

```
String s1 = "Hello ";
String s2 = s1 + "world";
String s3 = "Hello world";
if (s2 == s3) {
    println("Equal");
} else {
    println("Not Equal");
}
```

Überraschenderweise wird hier "Not Equal" ausgegeben. Dies hat damit zu tun, dass wir die *equals()* Methode verwenden müssen wenn wir zwei Strings vergleichen wollen, also:

```
String s1 = "Hello ";
String s2 = s1 + "world";
String s3 = "Hello world";
if (s2.equals(s3)) {
    println("Equal");
} else {
    println("Not Equal");
}
```

Warum das so ist werden wir etwas später herausfinden.

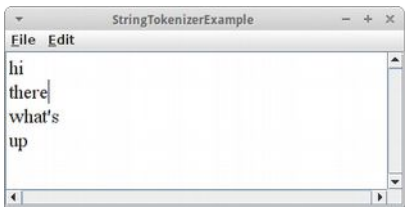
Übung: Einen String auseinanderschneiden

Als letzte kleine Übung zu Strings wollen wir einen String auseinanderschneiden. Dazu verwenden wir die Methoden *indexOf()* und *substring()*.

```
String s1 = "Hello world";
int posSpace = s1.indexOf(' ');
String s2 = s1.substring(0, posSpace);
String s3 = s1.substring(posSpace);
println(s2);
println(s3);
```

Wir finden die Position des Leerzeichens mit Hilfe der *indexOf()* Methode. Dann schneiden wir den vorderen Teil mit der *substring()* Methode ab. Wichtig sind hier zwei Dinge: das erste Zeichen ist an Position 0, d.h. wie üblich fangen wir bei 0 an zu zählen, und das Zeichen an der Position *posSpace* wird nicht mit abgeschnitten. Schließlich schneiden wir den Rest ab, also s3 ist alles ab Position *posSpace*.

StringTokenizer



Das Auseinanderschneiden von Strings ist etwas das wir so häufig tun werden, dass es dafür eine eigene Klasse gibt, den *StringTokenizer*.

```
    String sentence = "hi there what's up?";
    StringTokenizer st = new
StringTokenizer( sentence, " .,?" );
    while ( st.hasMoreTokens() ) {
        println( st.nextToken() );
    }
```

Zunächst initialisieren wir den StringTokenizer indem wir ihm sagen, was er auseinander schneiden soll (sentence), und wie, also, welche Trennzeichen er verwenden soll (' ','.',',' und '?'). Der StringTokenizer zerlegt den String in Tokens, also eigentlich Wörter. Um an diese Wörter zu gelangen, fragen wir den StringTokenizer ob er Tokens hat, also wir rufen die Methode *hasMoreTokens()* auf. Falls ja, bitten wir den StringTokenizer mittels *nextToken()* uns doch das nächste Token zu geben. Und das machen wir solange, bis alle Tokens aufgebraucht sind.

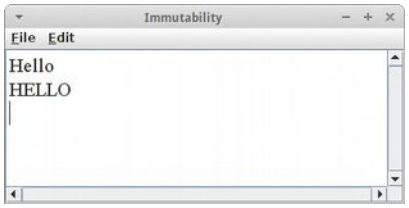
Immutability

Etwas das manchmal zu Verwirrungen führt, ist die *Immutability* von Strings. Immutability heißt Unveränderlichkeit, also soviel wie ein String kann nicht verändert werden. Am besten wir sehen uns ein Beispiel an:

```
String s1 = "Hello";
s1.toUpperCase();
println( s1 );
```

Was wird auf der Konsole ausgegeben? Es wird "Hello" ausgegeben, und nicht wie man vielleicht hoffen würde "HELLO". Das hat damit zu tun, dass Strings unveränderlich sind.

Es gibt aber einen Trick, wie wir Strings trotzdem verändern können: wir weisen ihnen einfach einen neuen Wert zu:




```
String s1 = "Hello";
s1 = s1.toUpperCase();
println( s1 );
```

Klassen

Wir haben jetzt schon viel von Klassen gehört, und auch schon viele Klassen verwendet, z.B. die Karel Klassen, aber auch die Graphik Klassen wie GRect und GOval. Zuletzt haben wir die String Klasse und die StringTokenizer Klasse verwendet. Die Frage die sich stellt, können wir auch eigene Klassen machen?

Natürlich, und es ist gar nicht mal so schwer. Als erstes Beispiel betrachten wir mal einen Studenten aus der Sicht einer Hochschule. Welche Information müsste denn eine Hochschule über einen Studenten wissen? Wenn wir uns auf das Wesentliche beschränken, dann sind das Name, Matrikelnummer und Leistungspunkte (ECTS). Wir nehmen also diese Eigenschaften (Attribute) eines Studenten und fassen sie zusammen zu einer Klasse:

```
public class Student {
    private String name;
    private int id;
    private double credits;
}
```

Im Prinzip ist also eine Klasse nichts anderes als ein Datencontainer. Jede Klasse sollte in ihrer eigenen Datei gespeichert werden, der Name der Datei muss genauso lauten wie der Name der Klasse.

SEP: Klassennamen sollten immer mit Großbuchstaben beginnen und der CamelCase Konvention folgen.

SEP: Information Hiding: Die Sichtbarkeit von Attributen sollte immer *private* sein.

Konstuktor

Allerdings sind Klassen 'intelligente' Datencontainer. Soll heißen, dass sie die Daten nicht nur enthalten und bündeln, sondern auch initialisieren, verwalten und ändern. Sie tun also auch etwas mit den Daten, und dafür haben Klassen Methoden. Die wichtigste Methode ist der *Konstuktor*, der initialisiert die Daten.

```
public Student(String n, int i, double c) {
    name = n;
    id = i;
    credits = c;
}
```

Der Konstuktor hat immer den gleichen Namen wie die Klasse, manchmal kann es mehr als einen Konstuktor geben, sie unterscheiden sich dann durch ihre Parameter. Ein Konstuktor hat nie einen Rückgabewert, also ein return.

So, jetzt haben wir eine Klasse, was machen wir damit? Wir benutzen sie wie jede andere Klasse. Dazu schreiben wir ein ConsolenProgram in dem wir mal eine Instanz einer Student Klasse erzeugen:

```
public class University extends ConsoleProgram {
    public void run() {
        Student fritz = new Student("Fritz", 12345, 0.0);
    }
}
```

Wir erzeugen also eine neue Instanz (*new*) der Klasse Student. Diese Instanz (auch Objekt genannt) soll *fritz* heißen. Um das Objekt zu erzeugen müssen wir den Konstruktor mit "new" aufrufen. Darin initialisieren wir die Instanzvariable *name* auf den Wert "Fritz", die Instanzvariable *id* auf den Wert 12345, und die Instanzvariable *credits* auf den Wert 0.0.

Methoden

Viel anfangen können wir mit dem *fritz* noch nicht, weil er nämlich nichts zu tun hat, also keine Methoden hat. Das können wir aber schnell ändern in dem wir "getter" Methoden hinzufügen:

```
public String getName() {
    return name;
}

public int getId() {
    return id;
}

public double getCredits() {
    return credits;
}
```

Jetzt können wir auf die Daten der Klasse lesend zugreifen. Um die Daten der Klasse aber zu verändern brauchen wir noch "setter" Methoden.

```
public void setName(String n) {
    name = n;
}

public void incrementCredits(double c) {
    if (c >= 0) {
        credits += c;
    }
}
```

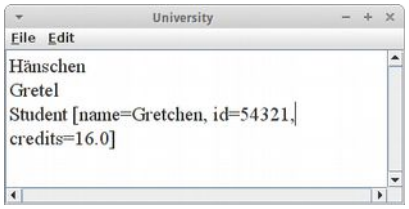
Nicht immer wollen wir dem Benutzer unserer Klasse erlauben Daten zu ändern. Z.B., kann ein Student zwar seinen Namen ändern, seine Matrikelnummer kann er aber nicht ändern. Deswegen gibt es keine *setId()* Methode. Auch soll ein Student nie negative Leistungspunkte erhalten können. Deswegen haben wir anstelle einer *setCredits()* Methode die *incrementCredits()*: diese stellt sicher, dass Leistungspunkte nur vermehrt werden können.

Abschließend, ist es noch üblich jeder Klasse eine *toString()* Methode zu geben:

```
public String toString() {
    return "Student [name=" + name + ", id=" + id
            + ", credits=" + credits + "]";
}
```

Die Aufgabe der *toString()* Methode ist es Informationen über das Objekt, also in der Regel die Werte der Attribute, auszugeben.

Übung: Hansel und Gretel



Die Klasse Student ist jetzt einsatzbereit und als Beispiel schauen wir uns mal die beiden Studenten Hansel und Gretel an der Brüder Grimm Berufsakademie an:

```
public class University extends ConsoleProgram {

    public void run() {
        Student hansel = new Student("Hänschen", 12345, 0.0);
        println(hansel.getName());

        Student gretel = new Student("Gretel", 54321, 11.0);
        println(gretel.getName());
        gretel.setName("Gretchen");
        gretel.incrementCredits(5);

        println(gretel.toString());
    }
}
```

Wrapper

Es ist nicht selten, dass man einen String in eine Zahl umwandeln muss. Dafür gibt es die Wrapper Klassen *Integer* und *Double*. Wenn man z.B. aus dem String "42" die Zahl 42 machen möchte, dann verwendet man die parseInt() Methode der Integer Klasse:

```
String fourtyTwo = "42";
int x = Integer.parseInt(fourtyTwo);
```

Umgekehrt um aus einer Zahl einen String zu erzeugen kann man auch wieder die Integer Klasse verwenden,

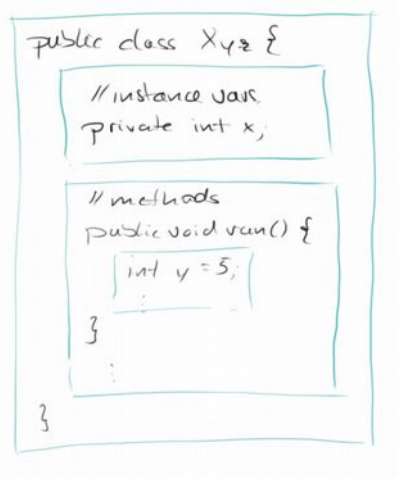
```
int x = 42;
String fourtyTwo = Integer.toString(x);
```

oder man behilft sich mit einem kleinen Trick:

```
String fourtyTwo = "" + x;
```

Es gibt für jeden primitiven Datentyp, also z.B. int, double, boolean und char, eine Wrapper Klasse. Es gibt sogar eine Wrapper Klasse für *void*, allerdings ist das wahrscheinlich die nutzloseste Klasse der Welt.

Review



In diesem Kapitel haben wir uns hauptsächlich mit Strings und mit Klassen beschäftigt. Wir haben das erste Mal eine eigene Klasse geschrieben. Dabei haben wir gesehen, dass Klassen immer wie folgt aufgebaut sind:

```
public class Name {
    // variables
    // methods
}
```

d.h. eine Klasse hat Variablen und Methoden. Manchmal nennen wir die Variablen auch Instanzvariablen, Eigenschaften einer Klasse oder auch Attribute einer Klasse. Bei den Methoden haben wir gesehen, dass diese immer etwas tun, und dass es eine spezielle Methode gibt, den Konstruktor.

Außerdem haben wir die folgenden Themen behandelt:

- Character
- String
- StringTokenizer
- Wrappers

Das wichtigste in diesem Kapitel waren aber die Klassen.

Projekte

Die Projekte in diesem Kapitel beschäftigen sich hauptsächlich mit Strings und dem Schreiben von einfachen Klassen.

Student

In dem Beispiel zu Student stellt sich die Frage warum die Instanzvariablen *private* waren? Betrachten wir das Problem einmal von der anderen Seite. Nehmen wir an Instanzvariablen wären *public*: könnte man dann verhindern, dass ein Student negative *credits* bekommt, oder dass seine *id* geändert werden kann?

ASCII

Um Zeichen (chars) zu speichern verwendet der Computer Zahlen. Deswegen kann man chars addieren, subtrahieren und man kann sie sogar in Bedingungen und Schleifen verwenden. Z.B. um festzustellen ob ein char ein Großbuchstabe ist kann man folgende Bedingung verwenden:

```
if ( c >= 'A' && c <= 'Z' ) { ... }
```

Oder um aus einem Großbuchstabem einen Kleinbuchstaben zu machen, hilft folgender Trick:

```
char gross = 'D';
char klein = gross - 'A' + 'a';
```

Man kann auch aus chars ints machen und umgekehrt via *Cast* bzw. Typumwandlung

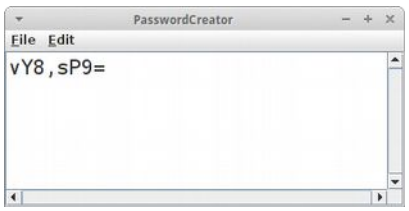
```
char c = 'A';
int ascii = (int) c;
char d = (char) ascii;
```

Mit diesem Wissen wollen wir die folgenden Methoden schreiben:

- isUpperCase(char c)
- toUpperCase(char c)
- printASCIITable()

Bei letzterer genügt es die ASCII Zeichen zwischen 32 und 128 auszugeben, da die ersten 31 ASCII Zeichen teilweise nicht druckbar sind.

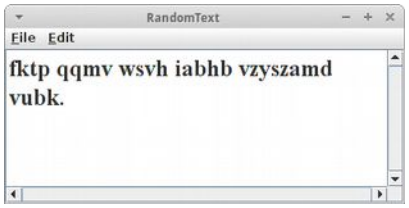
PasswordCreator



Mit unserem Wissen über Strings können wir jetzt einen Passwort Generator schreiben. Ein gutes Passwort soll mindestens 8 Zeichen lang sein, und mindestens je einen Kleinbuchstaben, einen Großbuchstaben, eine Ziffer und ein Symbol enthalten. Man könnte also vier Strings definieren die die möglichen Zeichen enthalten, und dann mit dem RandomGenerator daraus z.B. je zwei zufällig auswählen und zu einem Passwort zusammensetzen:

```
String small = "abcdefghijklmnopqrstuvwxyz";
password += small.charAt(rgen.nextInt(small.length()));
```

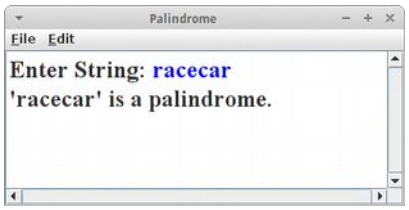
RandomText



Für Testzweck ist es manchmal hilfreich Zufallstext erzeugen zu können. Das ist gar nicht so schwer. Wir beginnen wieder mit dem Top-Down Ansatz. Nehmen wir an wir hätten eine Methode *createRandomSentence()* die einen Zufallssatz generiert. Dann könnten wir einen Zufallstext erzeugen, indem wir einfach mehrere Zufallssätze erzeugen. Die Methode *createRandomSentence()* wiederum ist einfach wenn wir annehmen es gäbe eine Methode namens *createRandomWord()*, denn dann würden wir einfach irgendwo zwischen drei und 5 Wörter nehmen, Lehrzeichen dazwischen setzen und einen Punkt am Ende hinzufügen. Und auch die *createRandomWord()* Methode ist nicht so schwer, denn Wörter bestehen aus Kleinbuchstaben, und haben in etwa 3 bis 8 Buchstaben, also

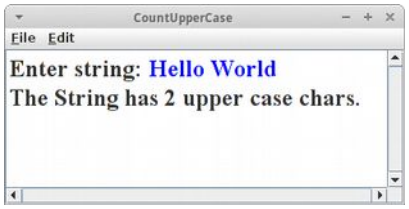
```
private String createRandomWord() {
    String word = "";
    int nrOfCharsInWord = rgen.nextInt(3, 8);
    for (int i = 0; i < nrOfCharsInWord; i++) {
        word += (char)('a' + rgen.nextInt(26));
    }
    return word;
}
```

Palindrome



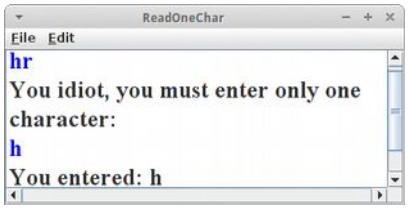
Worte die gleich bleiben wenn man sie umkehrt, wie z.B. "racecar" oder "rentner", nennt man auch Palindrome. Wir wollen eine Methode namens *isPalindrome(String s)* schreiben, die feststellt ob ein String ein Palindrom ist. Dazu schreiben wir eine Methode *reverse()* die einen gegebenen String umdreht, und dann vergleichen wir mittels *equals()* den Original mit dem umgekehrten String. Sind beide gleich, handelt es sich um ein Palindrom.

CountUpperCase



Hier sollen wir eine Methode schreiben, die die Großbuchstaben in einem String zählt. Das kann z.B. ganz nützlich sein wenn man bestimmen will ob ein bestimmter Text deutsch oder englisch ist: englischer Text hat im Durchschnitt weniger Großbuchstaben. Dazu können wir entweder die Character Klasse verwenden oder die Methode die wir oben selbst geschrieben haben.

ReadOneChar

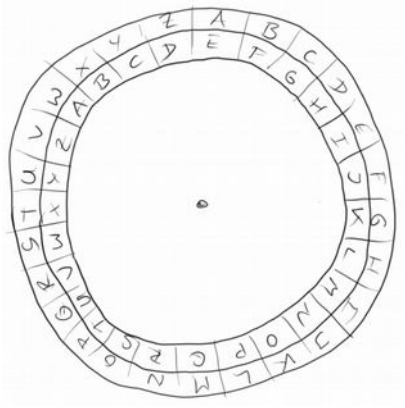


Wir möchten, dass der Nutzer nur einen Buchstaben eingeben kann. Er soll also nicht 0 oder mehrere, sondern genau einen Buchstaben eingeben. Dazu wollen wir eine Methode *readOneChar()* schreiben, die einen char als Rückgabewert liefert.

Es gibt die *readString()* Methode zum Einlesen von Strings. Wir benutzen also den Loop-and-a-Half, und benutzen als Abbruchkriterium, dass der eingelesene String die Länge eins haben soll.

```
while (true) {
    s = readLine();
    if (s.length() == 1)
        break;
    println("Please, enter only one character:");
}
```

Encrypt



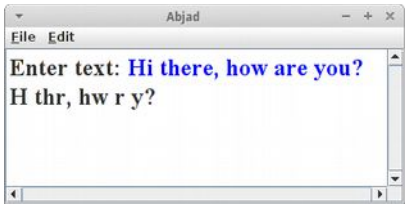
Eine andere interessante Anwendung, die wir jetzt ganz einfach realisieren können, ist ein kleines Verschlüsselungsprogramm. Wir verwenden dafür die sogenannte Caesar Cipher, die bereits Julius Caesar verwendet haben soll. Bei der Caesar Cipher [2] werden die Buchstaben in einem gegeben Text einfach um eine feste Anzahl, den Schlüssel, verschoben. Beträgt der Schlüssel z.B. vier, dann wird aus einem 'a' ein 'e', aus einem 'b' ein 'f', usw. Zum Entschlüsseln macht man den Vorgang einfach rückgängig.

Um das umzusetzen, schreiben wir zwei Methoden, *encrypt()* und *decrypt()*, die eine verschlüsselt, die andere entschlüsselt. Beide nehmen einen String *text* und einen int *key* als Parameter. Die eine gibt den verschlüsselten Text als String zurück, die andere den entschlüsselten.

Hinweise: damit das Ganze nicht zu kompliziert wird, macht es Sinn den Text erst einmal in Kleinbuchstaben umzuwandeln bevor man mit der Verschlüsselung anfängt. Und der Restwert Operator '%' ist hier sehr praktisch:

```
char c = 'a';
int d = c - 'a';
int e = d + key;
int f = e % 26;
char g = (char) ( f + 'a');
```

Abjad



Um etwa 1500 v. Chr. entwickelten die Phönizier die erste Alphabetschrift, eine linksläufige Konsonantenschrift. Unter Konsonantenschriften versteht man Schriftsysteme, in denen nur Zeichen für Konsonanten verwendet werden [2].

Um zu zeigen, dass eine solche Schrift durchaus lesbar sein kann, haben Archäologen uns beauftragt ein ConsoleProgramm zu schreiben, das aus einem gegebenen Text alle Vokale entfernt. Wir können wie folgt vorgehen:

- mittels readLine() bitten wir den Benutzer einen normalen Text einzugeben,
- diesen durchsuchen wir dann nach Vokalen und entfernen diese,
- das Resultat geben wir dann mittels println() aus.

Idealerweise erinnern wir uns an den Top-Down Ansatz, d.h. wir sollten vielleicht davon ausgehen, dass es die Methoden *removeVowels()* und *isVowel()* gibt, und diese dann später implementieren. Man könnte auch die *replace()* Methode der String Klasse verwenden, aber dazu mehr im nächsten Projekt.

Franconian

Die Sprache der Franken, auch 'lingua franca' genannt, ist zu unrecht in Vergessenheit geraten. Die 'lingua franca' (italienisch für: „fränkische Sprache“) ist eine romanisch basierte Pidgin-Sprache. Pidgin-Sprache oder Pidgin bezeichnet eine reduzierte Sprachform, die verschieden sprachigen Personen zur Verständigung dient [3].

Um also zur Verständigung der Völker beizutragen, sollen wir ein Deutsch-Fränkisches Übersetzungsprogramm schreiben. Im Fränkischen finden u.a. folgende lautlichen Vereinfachungen statt (auch als „binnendeutsche Konsonantenschwächung“ bekannt):

- t -> d
- p -> b
- k -> g

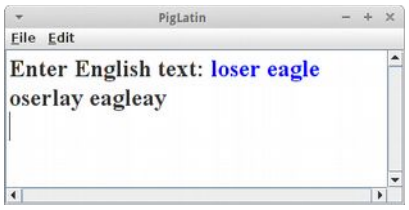
So wird beispielsweise aus dem 'Politiker' der 'Bolidiger'.

Wir schreiben also eine Methode *translateGermanToFranconian()*, die einen String als Parameter hat mit dem deutschen Text, und einen String als Rückgabewert hat, welcher die übersetzte fränkische Version liefert. Dazu könnte man die *replace()* Methode der String Klasse verwenden:

```
String german = "politiker";
String franconian = old.replace('t','d');
...
```

Anmerkung: In der String Klasse [4] gibt es auch die Methoden replaceAll(), replaceFirst() und split(). Diese sollten wir allerdings erst verwenden wenn wir wissen was "Reguläre Ausdrücke" sind (regular expressions).

PigLatin

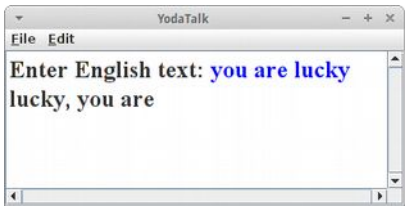


Pig Latin [5] ist eine Geheimsprache für Kinder die ganz einfache Regeln hat:

- beginnt ein Wort mit einem Konsonanten, dann wird der Konsonant ans Ende des Wortes verschoben und die Silbe "ay" angehängt. Also aus "loser" wird "oserlay" oder aus "button" wird "uttonbay",
- beginnt ein Wort mit einem Vokal, dann wird nur die Silbe "ay" angehängt. Also aus "eagle" wird "eagleay" und aus "america" wird "amercaay".

Wir sollen also auch hier wieder ein Konsolenprogramm schreiben, dass den Benutzer nach einem englischen Satz fragt und diesen dann ins Pig Latin übersetzt. Auch hier sollten wir wieder an den Top-Down Ansatz denken.

YodaTalk



In der Typologie von Sprachen [6], kommt die Satzstruktur Subjekt-Verb-Objekt (SVO), also das Subjekt an erster Stelle, das Verb an zweiter und das Objekt an dritter steht, sehr häufig vor. Circa 75% aller Sprachen der Welt folgen diesem oder einem sehr ähnlichen Muster. Wir wissen ja, dass Yoda vom Sumpfplaneten Dagobah kommt und dort ist der bevorzugte Satzbau Objekt-Subjekt-Verb (O,SV). Also, zum Beispiel, aus dem englischen Satz:

You are lucky.

wird auf "Yodish":

Lucky, you are.

Um die interplanetaren Kommunikation zu vereinfachen, ist es nun unsere Aufgabe, ein ConsoleProgramm zu schreiben, das einen gegebenen Text aus dem Englischen ins Yodische übersetzt. Wir können davon ausgehen, dass jeder eingegebene Satz immer aus drei Worten besteht, immer in der Reihenfolge SVO.

Wir könnten folgendermaßen vorgehen:

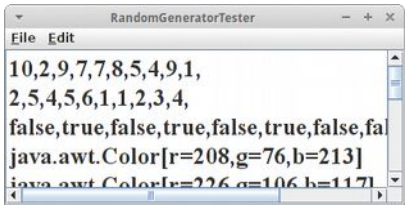
- wir bitten den Benutzer einen englischen Text einzugeben
- dann identifizieren wir die drei Elemente Subjekt, Verb und Objekt mit Hilfe eines StringTokenizer
- mit Hilfe von println(), geben wir dann das übersetzte Yodish in der Form Objekt, Subjekt Verb aus.

Natürlich verwenden wir wieder den Top-Down Ansatz und implementieren eine Methode namens *translateFromEnglishToYodish()*.

ELIZA

Um zu sehen was man mit Strings alles machen kann, schauen wir uns mal die Java Version des Programms ELIZA an. ELIZA simuliert einen Psychotherapeuten der uns Fragen stellt und mit dem wir dann eine kleine Konversation führen können. Man muss bedenken, dass ELIZA bereits 1966 von Joseph Weizenbaum programmiert wurde [7]. Es war das erste Programm das zeigte, dass man mit Computern auch mit normaler Sprache kommunizieren kann. Moderne Version von ELIZA sind Siri und die Telemarketing-Software Samantha, die kategorisch abstreitet, eine Maschine zu sein [8].

RandomGenerator



Wir wollen unsere eigene *RandomGenerator* Klasse schreiben. Dass ist gar nicht so schwer, wenn wir wissen, dass es im Standard Java eine Methode namens Math.random() gibt. Diese Methode gibt eine Gleitkommazahl zwischen 0 und 1, einschließlich der 0, aber ausschließlich der 1. Wenn wir damit eine Zahl zwischen 1 und 6 erzeugen wollen, dann geht das so:

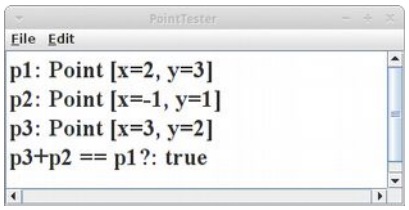
```
int diceRoll = 1 + (int)(Math.random() * 6);
```

Mit diesem Wissen wollen wir die Klasse RandomGenerator schreiben mit den Methoden nextInt(int a, int b), nextInt(int b), nextBoolean() und nextColor(). In Zukunft können wir jetzt immer unsere eigene Klasse anstelle der ACM Klasse verwenden.

Counter

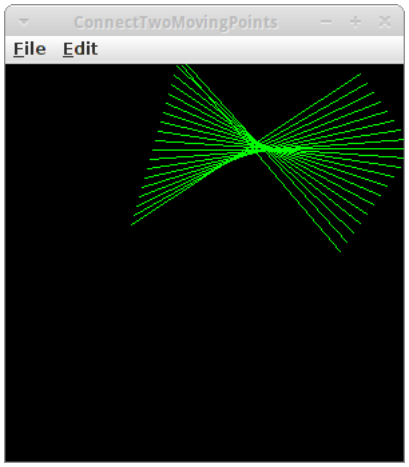
Wir wollen eine Klasse *Counter* schreben, die als Zähler fungierten kann. Die Klasse soll eine private Instanzvariable namens *count* haben. Im Konstruktor soll diese Variable auf den Wert 0 gesetzt werden. Dann soll es eine Methode namens *getValue()* geben, die einfach den momentanen Wert der Variable *count* zurückgibt. Und es soll eine Methode namens incrementCounter*()* geben, die den Wert der Variable *count* um eins erhöht.

Point



Als nächstes wollen wir eine Klasse *Point* schreiben. Diese soll einem Punkt in zwei dimensionalen Raum entsprechen, also eine x und eine y Koordinate haben. Die Klasse soll einen Konstuktor *Point(int x, int y)*, sowie die Methoden getX(), getY(), move(int dx, int dy), equals(Point p), add(Point p) und toString() haben.

ConnectTwoMovingPoints



Unsere Klasse Point scheint so für sich alleine relative langweilig. Wenn sich aber zwei Punkte bewegen, und man diese mit einer farbigen Linie verbindet, dann sieht das schon sehr hübsch aus.

Wir beginnen mit der Klasse *Point*, die allerdings etwas andere Anforderungen hat, als die vom letzten Projekt. Sie soll möglichst einfach sein. Sie soll vier Instanzvariablen haben, nämlich *x*, *y*, *vx* und *vy*. Alle vier sollen *public* sein. Der Konstruktor soll keine Parameter haben, und er soll die Instanzvariablen mit zufälligen Werten initialisieren. Die Klasse soll nur eine Methode haben:

```
public void move() {
    x += vx;
    y += vy;
}
```

Diese Klasse wollen wir in unserem GraphicsProgram *ConnectTwoMovingPoints* verwenden:

```
public void run() {
    Point p1 = new Point();
    Point p2 = new Point();

    while (true) {
        p1.move();
        p2.move();
        checkForCollisionWithWall(p1);
        checkForCollisionWithWall(p2);
        drawConnection(p1, p2);
        pause(DELAY);
    }
}
```

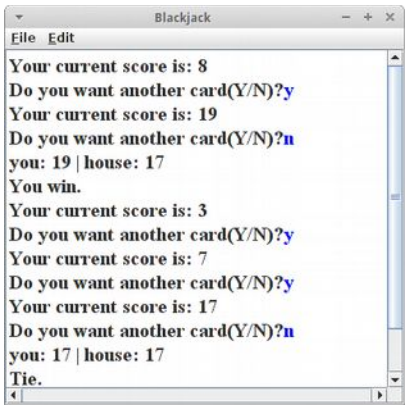
Wir erzeugen also zwei Punkte, *p1* und *p2*. In unserem GameLoop, bewegen wir die beiden Punkte, schauen nach ob die Punkte noch im Spielfeld sind, und dann verbinden wir die beiden Punkte mit einer Linie. Das ist eigentlich ganz einfach. Allerdings wenn man immer nur zehn Linien anzeigen möchte, und die älteren wieder löschen möchte, benötigt man Arrays (nächstes Kapitel).

Noch eine kleine Feinheit in dem Program: wenn wir uns die Definition der Klasse *Point* ansehen,

```
public class ConnectTwoMovingPoints extends GraphicsProgram {

    ...

    class Point {
        ...
    }
}
```

dann stellen wir fest, dass dies innerhalb der Klasse *ConnectTwoMovingPoints* geschieht. Man nennt die Klasse Point daher auch eine *lokale* Klasse, weil sie nur lokal innerhalb der Klasse ConnectTwoMovingPoints existiert. Es gibt ein paar wenige Fälle wie hier, wo so etwas Sinn macht. ansonsten macht man so etwas aber eher selten.

Blackjack

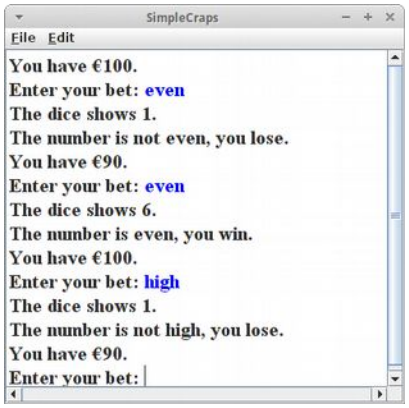


Laut Wikipedia ist "Black Jack das meistgespielte Karten-Glücksspiel, das in Spielbanken angeboten wird" [9]. Wir wollen eine etwas einfachere Version dieses Spiels implementieren.

Anstelle von Karten verwenden wir einfach Zahlen, und zwar Zahlen zwischen 1 und 11. Der Computer spielt den Croupier und beginnt indem er eine Zufallszahl zwischen 17 und 25 erzeugt. Dann ist der Spieler an der Reihe. Dieser fängt mit einer Karte an, also eine Zufallszahl zwischen 1 und 11. Er kann dann entscheiden ob er noch eine Karte möchte. Falls ja, wird wieder eine Zufallszahl zwischen 1 und 11 erzeugt und zur momentanen "Hand" hinzuaddiert. Wenn der Spieler keine neue Karte mehr haben möchte, wird die "Hand" des Spielers mit der des Computers verglichen.

Gewonnen hat derjenige der 21 Punkte oder weniger hat und mehr als der andere. Ansonsten ist es ein Unentschieden.

SimpleCraps



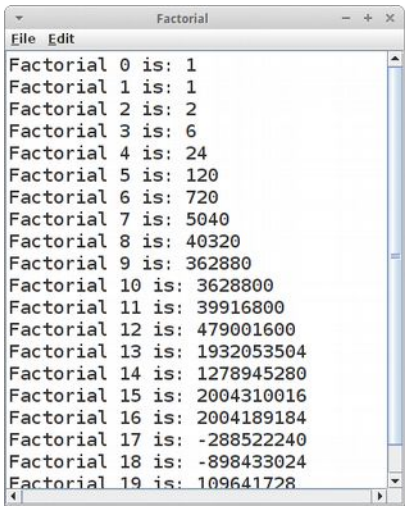
"Craps bzw. Craps shooting oder Seven Eleven ist ein Würfelspiel, das sich vor allem in den USA großer Beliebtheit erfreut." [10]

Wir werden eine einfachere Version von Craps implementieren: Bei uns gibt es nur einen ganz normalen Würfel. Der Spieler beginnt mit einem Guthaben von 100 Euro. In jeder Runde werden 10 Euro gesetzt und der Spieler auf eines der folgenden Resultate setzen:

- odd
- even
- high (4,5,6)
- low (1,2,3)

Das Spiel ist beendet, wenn kein Guthaben mehr vorhanden ist.

Factorial



Nicht alle Berechnungen die ein Computer macht sind richtig. Am besten wir sehen uns mal ein Beispiel an.

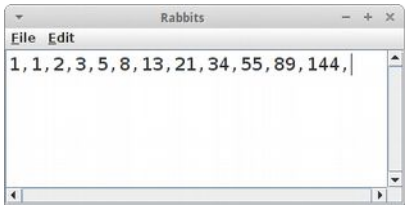
Die Fakultät einer Zahl ist das Produkt aller natürlichen Zahlen kleiner und gleich dieser Zahl. Z.B. ist die Fakultät von 3:

```
3! = 1 * 2 * 3 = 6
```

Wir wollen also eine Methode namens *calculateFactorial(int n)* schreiben, die die Fakultät der Zahl *n* berechnet. Und damit wollen wir dann die Fakultäten der Zahlen von 1 bis 20 ausgeben. Fakultäten berechnet man am besten mit einer *for* Schleife.

Die Fakultäten haben die Eigenschaft, dass sie sehr schnell sehr groß werden. Und das führt zu einem Problem, denn Computer können nicht so gut mit großen Zahlen rechnen!

Rabbits

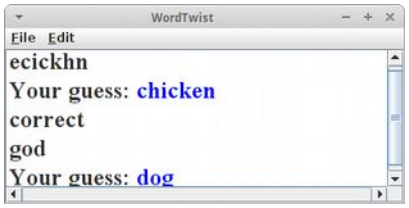


Jeder der schon mal Kaninchen hatte weiß, dass diese eine interessante Eigenschaft haben: sie vermehren sich und zwar rasant. Wir wollen also ein Programm schreiben dass berechnet wie sich unsere Kaninchen-Population über die Monate entwickelt. Wir folgen dazu dem Modell von Fibonacci [12]:

- "Jedes Paar Kaninchen wirft pro Monat ein weiteres Paar Kaninchen.
- Ein neugeborenes Paar bekommt erst im zweiten Lebensmonat Nachwuchs (die Austragungszeit reicht von einem Monat in den nächsten).
- Die Tiere befinden sich in einem abgeschlossenen Raum, sodass kein Tier die Population verlassen und keines von außen hinzukommen kann."

Wenn wir die Simulation richtig schreiben, dann müsste dabei die Fibonacci-Folge, also 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... herauskommen.

WordTwist



In dem Spiel *WordTwist* geht es darum ein Wort in dem einige Buchstaben vertauscht wurden, wiederzuerkennen. Also z.B. sollen wir erkennen, dass das Wort "nhickce" eigentlich aus dem Wort "chicken" entstanden ist.

Wir beginnen mit dem Ausgangswort, also z.B. "chicken", und vertauschen einige Buchstaben. Das zeigen wir dann dem Nutzer, und er soll dann raten welches das Ausgangswort war. Wenn wir wieder den Top-Down Ansatz verwenden, dann macht es Sinn zunächst eine Methode *scrambleWord(String word)* zu schreiben, die das Ausgangswort als Parameter nimmt, und das durchmischte Wort als Rückgabewert liefert. Wenn wir den Top-Down Ansatz weiterverfolgen, macht es auch Sinn eine Methode namens *randomSwap(String word)* zu schreiben, die einfach zwei zufällig gewählte Buchstaben in dem Wort vertauscht. Wenn wir diese Methode mehrmals aufrufen, dann kommt dabei ein gut durchmischtes Wort heraus.

Die Methode *pickGuessWord()* existiert bereits und liefert uns ein zufälliges Wort.

Challenges

Hangman

Galgenmännchen ist ein einfaches Buchstabenspiel [12]. Eigentlich ein grafisches Spiel, wollen wir hier eine Variante programmieren die textbasiert ist. Es beginnt damit, dass sich der Computer ein zufälliges Wort überlegt. Der Spieler darf dann einen Buchstaben raten. Das Programm zeigt dann an ob und an welcher Stelle im Wort dieser Buchstabe vorkommt. Das wird dann solange wiederholt, bis das Wort erraten wurde. Am Ende soll dann ausgegeben werden, wie viele Versuche notwendig waren um das Wort zu erraten. Dabei werden aber nur Fehlversuche gezählt.

Wir benötigen zwei Instanzvariablen,

```
private String guessWord;
private String hintWord;
```

wobei *guessWord* das Wort enthält das geraten werden soll (z.B. "mother") und *hintWord* enthält die Buchstaben die bisher korrekt geraten wurden, also am Anfang nur Striche (z.B. "------"). Wir müssen sowohl *guessWord* als auch *hintWord* initialisieren:

```
guessWord = pickGuessWord();
createHintWord();
```

wobei die Methode *pickGuessWord()* bereits existiert und wir die Methode *createHintWord()* aber noch schreiben müssen.

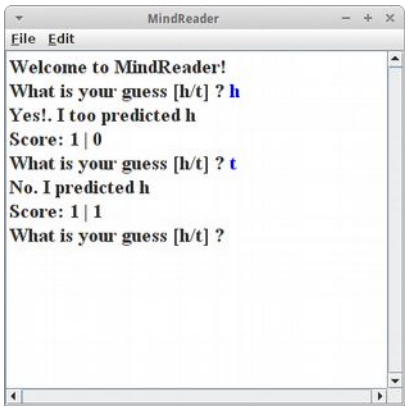
Danach beginnt der *GameLoop*: Im ersten Schritt zeigen wir dem Spieler das *hintWord* damit er eine Idee hat wie viele Buchstaben das Wort enthält. Dann bitten wir den Spieler einen Buchstaben einzugeben. Hier können wir die Methode *readOneChar()* aus dem Projekt ReadOneChar verwenden. Mittels *contains()*:

```
char c = readChar();
if (guessWord.contains("" + c)) {
    buildNewHintWord(c);
}
```

können wir feststellen, ob der neue Buchstabe in dem *guessWord* überhaupt vorkommt. Falls ja, müssen wir ein neues *hintWord* mittels der Methode *buildNewHintWord(c)* konstruieren. Diese Methode sollte den richtig geratenen Buchstaben an der korrekten Stelle im *hintWord* einfügen, und das neue *hintWord* sollte dann dem Spieler angezeigt werden.

Das Abbruchkriterium ist relativ einfach, wenn es keine Striche "-" mehr im *hintWord* gibt, dann hat der Spieler das Wort erraten. Alternativ könnte man natürlich auch *guessWord* und *hintWord* vergleichen.

MindReader



In diesem Spiel geht es um das Gedankenlesen. Es ist inspiriert vom Münzwurf, d.h. es gibt Kopf (heads) und Zahl (tails). Aber anstelle eine Münze zu werfen, wählt der Spieler einfach Kopf oder Zahl. Der Computer versucht nun vorherzusagen was der Spieler raten wird. War die Vorhersage des Computers richtig, erhält der Computer einen Punkt, ansonsten der Spieler. Wer als erstes 25 Punkte hat, hat gewonnen. Die Idee des Spiels basiert auf einem Handout von Professor Raja Sooriamurthi, der wiederum von Professor Gregory Rawlins inspiriert wurde [13].

Wenn wir den Top-Down Ansatz verwenden, dann ergibt sich folgende Grobstruktur:

- der Computer macht eine Vorhersage, entweder Kopf oder Zahl (computerMakePrediction())
- der Spieler trifft seine Auswahl, entweder Kopf oder Zahl (humanMakePick())
- die Vorhersage des Computers wird angezeigt (revealPrediction())
- die Computer-Vorhersage wird mit der Auswahl des Spielers verglichen und je nachdem ob der Computer richtig geraten hat oder nicht, erhält entweder der Computer einen Punkt oder der Spieler.

Das Ganze wird dann so lange wiederholt bis einer 25 Punkte erreicht hat.

Für das Spiel benötigen wir vier Instanzvariablen:

```
private char computerGuess;
private char humanGuess;
private int computerScore = 0;

private int humanScore = 0;
```

und zwar jeweils den Buchstaben den der Spieler eingegeben hat und den den der Computer vorhergesagt hat, sowie den Spielstand, also die Punkte des Spielers und die des Computers.

Die Vorhersage des Computers könnte man einfach zufällig machen, aber dann ist das Spiel eher langweilig. Viel interessanter wird es mit der bereits existierenden Klasse *MindReaderPredictor*:

```
predictor = new MindReaderPredictor();
```

Diese Klasse hat zwei Methoden, *makePrediction()* und *addNewGuess(char c)*. Die erste Methode versucht eine Vorhersage zu machen, gibt also 'h' oder 't' als Rückgabewert. Die zweite fügt eine neue Spielerauswahl zur Datenbank des Predictors hinzu, und erlaubt es dem Predictor bessere Vorhersagen zu machen. Der Predictor versucht also vom Spieler zu lernen. Der *predictor* sollte natürlich auch eine Instanzvariable sein.

Erweiterungen:

Man kann sich für das Spiel ganz viele Erweiterungen einfallen lassen. z.B.:

- könnte man mehrere Spiele gleich hintereinander spielen, dabei sollte aber der MindReaderPredictor nicht jedes mal neu initialisiert werden
- man könnte eine eigene MindReaderPredictor Klasse schreiben, eine ganz einfache Version würde einfach nur zufällig zwischen 'h' und 't' wählen

Fragen

1. Geben Sie ein Beispiel für eine Klasse und ein Beispiel für ein Objekt.

2. "arnold" ist eine "Schauspieler". Ist "arnold" ein Objekt oder eine Klasse?

3. Was ist normalerweise die Aufgabe eines Konstruktors?

4. Betrachten Sie die folgenden Klassen: SteepleChaserKarel, GRect, Karel, GOval, GObject. Welche ist die Superklasse von welcher?

5. Es ist guter Stil jeder Klasse eine toString() Methode zu geben. Was sollte die toString() Methode machen?

6. Wie konvertieren Sie einen String der eine Zahl enthält, z.B. "42", in einen int?

7. Strings sind IMMUTABLE, also unveränderlich, was bedeutet das?

8. Wenn man zwei Strings vergleichen möchte muss man etwas vorsichtig sein. Welcher Fehler tritt sehr häufig beim Vergleichen von Strings auf?

9. Die Klasse String hat viele Methoden. Die folgenden haben wir benutzt. Beschreiben Sie kurz, was jede dieser Methoden macht.
   substring()
   length()
   charAt()
   toLowerCase()
   indexOf()

10. Schreiben Sie Beispielcode der einen String umkehrt, also z.B. das Wort ”STRESSED” in das Wort “DESSERT” verwandelt. (Der reine Java Code genügt, eine Klassen oder Methoden Deklaration ist nicht notwendig)

11. Wofür verwendet man die StringTokenizer Klasse?

12. Die Klasse StringTokenizer nimmt einen String als Parameter und hat zwei Methoden namens hasMoreTokens() und nextToken(). Schreiben Sie Code der den Nutzer nach einem Satz fragt, und dann die einzelnen Worte des Satzes ausgibt, Zeile für Zeile.

13. Der nachfolgende Code hat Probleme. Mindestens fünf Code Konventionen wurden dabei verletzt. Listen Sie diese bitte auf. (1 Punkt)

```
public class factorialExample extends ConsoleProgram {

int j = 3;
private static final int maxnum = 10;

public void run() {
for (int i = 0; i < maxnum; i++) {
println(i + "! = " + factorial(i));
}
}

private int Factorial(int n) {
           int RESULT = 1;
           for (int i = 1; i <= n; i++)
           result *= i;
           return result;
       }
   }
```

Referenzen

Die Referenzen aus Kapitel 2 sind auch hier weiter wichtig. Außerdem enthält die Wikipedia zahlreiche Informationen.

[1] Caesar cipher, https://en.wikipedia.org/w/index.php?title=Caesar_cipher&oldid=702242426 (last visited Feb. 17, 2016).

[2] Abjad, https://en.wikipedia.org/w/index.php?title=Abjad&oldid=704692935 (last visited Feb. 17, 2016).

[3] Pidgin-Sprachen, https://de.wikipedia.org/w/index.php?title=Pidgin-Sprachen&oldid=147087583 (last visited Feb. 17, 2016).

[4] String (Java Platform SE 7) - Oracle Documentation, https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

[5] Pig Latin, https://de.wikipedia.org/w/index.php?title=Pig_Latin&oldid=149209295 (Abgerufen: 17. Februar 2016, 20:54 UTC).

[6] Sprachtypologie, https://de.wikipedia.org/w/index.php?title=Sprachtypologie&oldid=149705787 (Abgerufen: 17. Februar 2016, 21:10 UTC).

[7] ELIZA, https://en.wikipedia.org/w/index.php?title=ELIZA&oldid=704986757 (last visited Feb. 17, 2016).

[8] Telemarketing-Software Samantha, http://de.engadget.com/2013/12/11/audio-telemarketing-software-samatha-streitet-kategorisch-ab-e/

[9] Blackjack, https://en.wikipedia.org/wiki/Blackjack

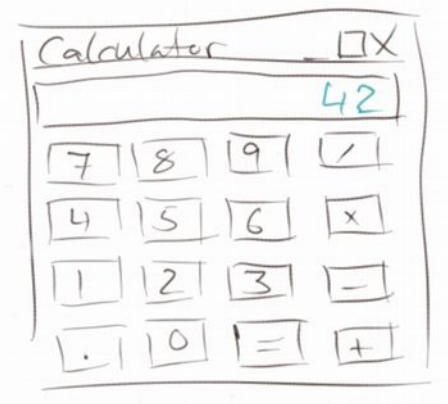
[10] Craps, https://de.wikipedia.org/wiki/Craps

[11] Fibonacci-Folge, https://de.wikipedia.org/wiki/Fibonacci-Folge

[12] Galgenmännchen, https://de.wikipedia.org/wiki/Galgenmännchen

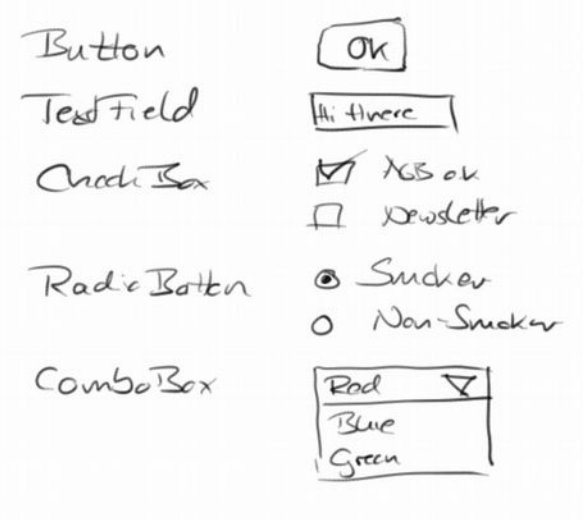
[13] Mind Reader: a program that predicts choices, http://nifty.stanford.edu/2007/raja-mindreader/

Swing



Programme mit grafischer Benutzeroberfläche (GUI) sind das Thema in diesem Kapitel. Ähnlich wie bei Grafikprogrammen gibt es vorgefertigte Komponenten mit denen wir dann die verschiedensten kleinen und großen Programme bauen. Das ist ein bisschen wie Lego. Zusätzlich werden wir hier noch ein wenig mehr über Instanzvariablen erfahren.

Grafische Benutzeroberfläche

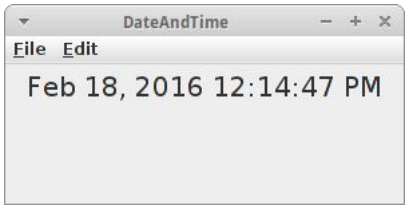


Die meisten Programme mit denen wir täglich zu tun haben, sind Programme mit einer grafischen Benutzeroberfläche, oder englisch *graphical user interface* (GUI oder UI). GUIs bestehen aus grafischen Elementen, die wir auch Widgets oder Interaktoren nennen, weil wir mit ihnen interagieren können. Bekannte Beispiele sind:

- Labels
- Buttons
- Textfields
- Checkboxes
- Radiobuttons
- Comboboxes

Im Folgenden werden wir einen nach dem anderen ausprobieren.

JLabel

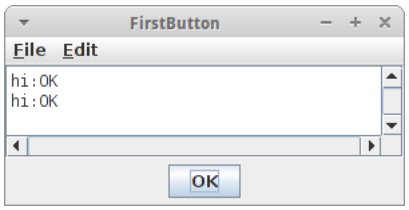


Wir beginnen wieder ganz einfach, mit dem JLabel. Es gibt praktisch keinen Unterschied zum GLabel. Er wird verwendet um Text in GUI Programmen darzustellen:

```
public class DateAndTime extends Program {

    public void init() {
        Date dt = new Date();
        JLabel fritz = new JLabel(dt.toLocaleString());
        fritz.setFont( new Font("SansSerif", Font.PLAIN, 20) );
        add( fritz, NORTH );
    }
}
```

Zu unseren bisherigen Programmen sehen wir zwei Unterschiede: wir schreiben jetzt "*extends Program*" anstelle von GraphicsProgram oder ConsoleProgram. Und wir verwenden die *init()* Methode anstatt der *run()* Methode. Das ist zwar nicht zwingend notwendig, aber guter Stil. Des weiteren sehen wir zum ersten Mal die Klasse *Date*. Die ist ganz praktisch wenn man Datum oder Uhrzeit benötigt. Und wir sehen die Klasse *Font*, mit der man den Font eines JLabels ändern kann.

JButton



Der JLabel ist nicht besonders interaktiv, und verdient den Namen eigentlich gar nicht. Anders ist das allerdings beim *JButton*, denn auf den kann man mit der Maus klicken. Wir fangen ganz einfach an:

```
public class FirstButton extends ConsoleProgram {
    public void init() {
        JButton btn = new JButton("OK");
        add( btn, SOUTH );
    }
}
```

Wir legen einen neuen JButton an, auf dem "OK" stehen soll. Den fügen wir dann im Süden hinzu. Unser FirstButton Programm ist wieder ein ConsoleProgram, das ist aber o.k. Wenn wir das Programm starten, können wir zwar auf den Knopf drücken, es passiert aber nichts.

Damit der Knopf richtig funktioniert, also interaktiv wird, müssen wir zwei Dinge tun:

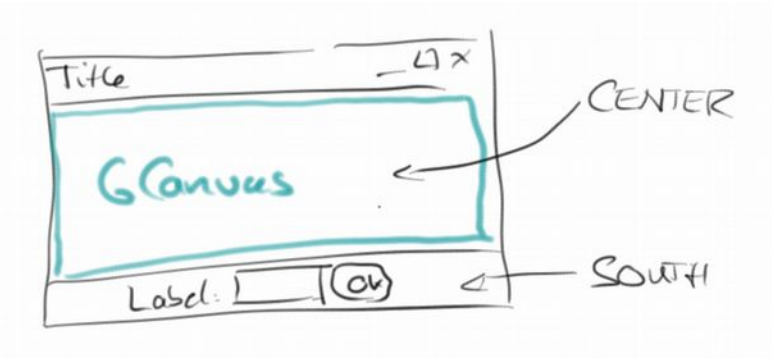
1. wir müssen am Ende von init() die Method *addActionListeners()* aufrufen und
2. wir müssen eine Methode namens *public void actionPerformed()* implementieren.

Der erste Schritt sagt dem Programm, dass wir informiert werden möchten wenn irgendein Knopf gedrückt wird. Und wie werden wir informiert? Man ruft uns an, allerdings nicht auf dem Telefon, sondern mit der Methode *actionPerformed()*. D.h., jedes Mal wenn jemand auf den Knopf drückt, dann wird diese Methode aufgerufen. Schauen wir uns das im Ganzen noch mal an:

```
public class FirstButton extends ConsoleProgram {

    public void init() {
        JButton btn = new JButton("OK");
        add( btn, SOUTH );

        addActionListeners();
    }

    public void actionPerformed( ActionEvent e ) {
        println("hi:" + e.getActionCommand());
    }
}
```

Jetzt wird in dem Konsolenteil unseres Programms jedes Mal "hi:" mit dem Namen des Knopfes ausgegeben, wenn wir mit der Maus auf den Knopf drücken.

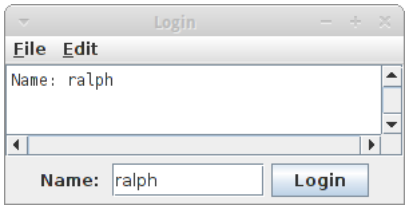
Regionen



In den zwei Beispielen oben, haben wir bereits zwei Regionen kennengelernt, insgesamt gibt es fünf: EAST, WEST, SOUTH, NORTH, und CENTER. Wir können unsere Widgets in jede dieser Regionen einfügen.

Eines was ein *ConsoleProgram* von einem *GraphicsProgram* und wiederum von einem *Program* unterscheidet, ist was in der CENTER Region ist. Beim *Program* ist da gar nichts. Beim *GraphicsProgram* ist da ein GCanvas eingefügt, und beim *ConsoleProgram* eine TextArea.

JTextField



Mittels des JTextField Widgets können wir Text und auch Zahlen einlesen. Im folgenden Beispiel möchten wir, dass sich die Nutzer mit ihrem Namen anmelden.

```
public class Login extends ConsoleProgram {
    private JTextField tf;

    public void init() {
        JLabel lbl = new JLabel("Name: ");
        add(lbl, SOUTH);

        tf = new JTextField(10);
        add(tf, SOUTH);

        JButton btn = new JButton("Login");
        add( btn, SOUTH );

        addActionListeners();
    }

    public void actionPerformed(ActionEvent e) {
```

```
            println("Name: " + tf.getText());
        }
    }
```

Wir legen einen JLabel, ein JTextField und einen JButton an. Im Konstruktor des JTextField sagen wir wie breit das Feld sein soll. Wichtig ist noch, dass das JTextField *tf* eine Instanzvariable ist. Wäre es nämlich eine lokale Variable, dann könnten wir nicht in der *actionPerformed()* Methode darauf zugreifen.

Ein kleine Verbesserung des Codes bringt die folgende Zeile

```
            tf.addActionListener(this);
```

die wir direkt vor dem add(tf) einfügen können. Diese Zeile bewirkt nämlich, dass es auch genügt einfach die "Enter" Taste zu drücken um sich einzuloggen.

Instanzvariablen

Wir haben zwar schon im letzten Kapitel kurz mal mit Instanzvariablen zu tun gehabt, aber gerade im obigen Beispiel haben wir gesehen wofür sie eigentlich gut sind: sie erlauben es zwischen verschiedenen Methoden Information auszutauschen. Bisher konnten wir dafür lediglich Parameter und Rückgabewerte verwenden.

Ein anderer Grund warum Instanzvariablen ganz praktisch sein können, hat damit zu tun, dass alle lokalen Variablen wieder gelöscht werden, wenn eine Methode verlassen wird. Also z.B. in der Methode *rollTheDie()*,

```
    private void rollTheDie() {
        RandomGenerator rgen = new RandomGenerator();
        int dieRoll = rgen.nextInt(1,6);
        println(dieRoll);
    }
```

wird bei jedem Aufruf ein neue Instanz des RandomGenerator erzeugt und dann beim Verlassen wieder gelöscht. Beides braucht Zeit. Wenn wir aber den RandomGenerator als Instanzvariable anlegen,

```
public class InstanceVariables extends ConsoleProgram {
    private RandomGenerator rgen = new RandomGenerator();

    private void rollTheDie() {
        int dieRoll = rgen.nextInt(1,6);
        println(dieRoll);
    }
}
```

dann müssen wir ihn nur einmal anlegen, und zusätzlich können wir ihn auch in anderen Methoden verwenden. Das ist viel resourcenschonender.

Instanzvariablen vs. Lokale Variablen

Es gibt also zwei Arten von Variablem: Instanzvariablen und lokale Variablen. Für Instanzvariablen gilt, dass sie in der Klasse deklariert werden und nicht in einer Methode, dass sie in der gesamten Klasse sichtbar sind, also von allen Methoden einer Klasse auf sie zugegriffen werden kann und dass sie solange leben wie das Objekt existiert.

Für lokale Variablen gilt umgekehrt, dass sie innerhalb einer Methode deklariert werden, dass nur innerhalb dieser Methode auf sie zugegriffen werden kann und dass sie beim Verlassen der Methode wieder gelöscht werden.

Nun stellt sich die Frage, wann verwende ich welche? Die Antwort ist in den meisten Fällen relativ einfach:

- wenn es sich um eine Berechnung handelt, die lokal in einer Methode ausgeführt werden kann, dann verwendet man lokale Variablen, z.B. die Umwandlung von Grad nach Fahrenheit ist eine lokale Berechnung
- wenn eine Information in mehreren Methoden verwendet wird, dann sollte man eine Instanzvariable nehmen
- wenn es sich um den internen Zustand eines Objektes handelt, dann sollte man eine Instanzvariable verwenden. Z.B. bei der Klasse Student waren der Name, die Id und die Credits interne Zustände.

SEP: Wenn möglich sollte man lokalen Variablen verwenden.

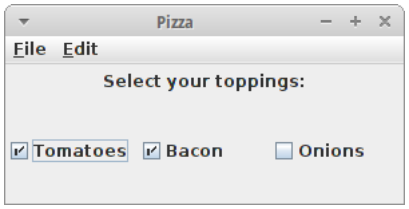
Übung: OKCancel

Als kleine Übung fügen wir noch einen *Cancel* Knopf zu unserem Programm hinzu. Wie können wir jetzt unterscheiden welcher der beiden Knöpfe denn gedrückt wurde? Hier ist manchmal sinnvoller die *getSource()* Methode des ActionEvents zu verwenden,

```
public void actionPerformed( ActionEvent e ) {
  if( e.getSource() == btnOK ) {
    ...
  }
}
```

um zwischen mehreren Knöpfen oder Widgets im Allgemeinen zu unterscheiden.

JCheckBox



Als nächstes Widget widmen wir uns den Checkboxes: Diese verwenden wir wenn wir mehrere Auswahlmöglichkeiten haben.

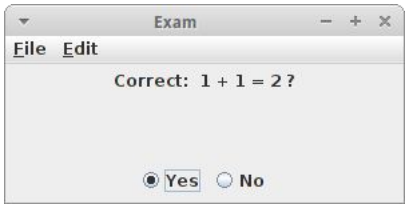
```
public void init() {
    JLabel lbl =
        new JLabel("Select your toppings:");
    add(lbl, NORTH);

    JCheckBox topping1 = new JCheckBox("Tomatoes");
    add(topping1, CENTER);
    JCheckBox topping2 = new JCheckBox("Bacon");
    add(topping2, CENTER);
    JCheckBox topping3 = new JCheckBox("Onions");
    add(topping3, CENTER);
}
```

In dem Pizza Beispiel soll es möglich sein alle möglichen Kombinationen von Toppings auszuwählen. Das kann man am besten mit Checkboxes erreichen. Um festzustellen welche Toppings ausgewählt wurden, gibt es die Methode *isSelected()*:

```
boolean b = topping1.isSelected();
```

Das müssten wir natürlich für alle drei Toppings machen.

JRadioButton



Radiobuttons werden verwendet wenn es darum geht Entscheidungen zu treffen. Betrachten wir das folgende Beispiel:

```
public void init() {
    JLabel lbl =
        new JLabel("Correct:  1 + 1 = 2 ?");
    add(lbl, NORTH);

    JRadioButton yes = new JRadioButton("Yes");
    yes.setSelected(true);
    add(yes, SOUTH);

    JRadioButton no = new JRadioButton("No");
    add(no, SOUTH);

    ButtonGroup happyGrp = new ButtonGroup();
    happyGrp.add(yes);
    happyGrp.add(no);
}
```

Wir haben hier zwei Radiobuttons, wobei der "yes" Button vorselektiert ist. Damit das Programm weiß welche Buttons zusammengehören, fügt man zusammengehörige Buttons in eine ButtonGroup zusammen. Das führt dann dazu, dass immer nur einer der Buttons ausgewählt sein kann. Will man wissen welchen Button der Nutzer gewählt hat, kann man das mittels:

```
boolean b = yes.isSelected();
```

feststellen.

JComboBox

Als Beispiel für die JComboBox schauen wir uns das Beispiel FavoriteColor an. Es geht darum eine Farbe aus einer Liste auszuwählen:

```
public class FavoriteColor extends ConsoleProgram {
    private JComboBox colorPicker;

    public void init() {
        colorPicker = new JComboBox();
        colorPicker.addItem("Red");
        colorPicker.addItem("White");
        colorPicker.addItem("Blue");
        add(colorPicker, SOUTH);

        addActionListeners();
    }

    public void actionPerformed(ActionEvent e) {
        println("Color:" + colorPicker.getSelectedItem());
    }
}
```

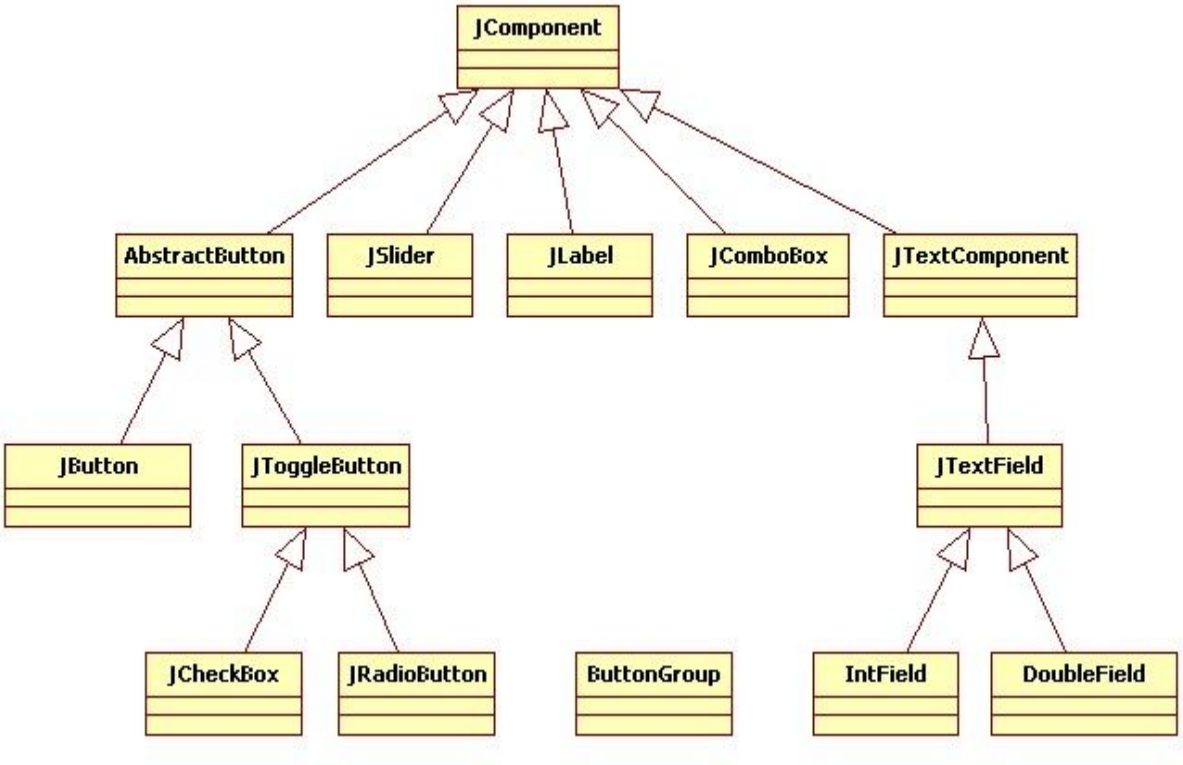
In der init() Methode initialisieren wir den *colorPicker* mit den voreingestellten Farben. Um festzustellen welche Farbe der Benutzer ausgewählt hat, gibt es die *getSelectedItem()* Methode. Damit wir aber auf diese in der *actionPerformed()* Methode zugreifen können, muss *colorPicker* eine Instanzvariable sein.

Anmerkung: wenn wir ganz genau das Verhalten der JComboBox beobachten, werden wir feststellen, dass sie ein paar klein Quirks hat. Kann man nix machen.

.

Swing Interactor Hierarchie

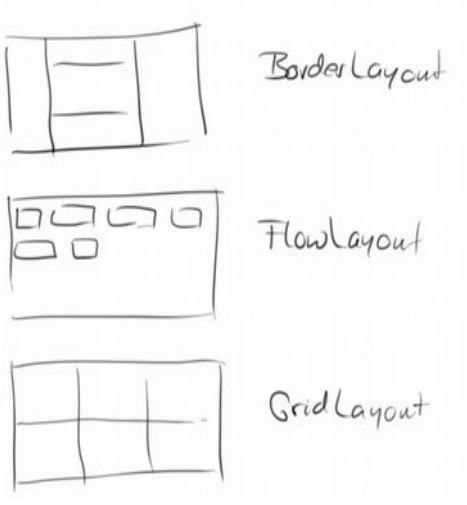
Ähnlich wie es bei den ACM Grafikklassen eine Hierarchie gibt, gibt es die auch bei den Swing Interactor Klassen. Die wichtigsten sind in dem nebenstehenden Diagramm zusammengefasst, es gibt aber noch weitaus mehr.



Layout

Wir haben bereits die Regionen EAST, WEST, SOUTH, NORTH, und CENTER kennengelernt. Diese sind eine Besonderheit des BorderLayouts. Es gibt neben dem BorderLayout noch einige andere. Es geht darum wie man mehrere Widgets auf dem Bildschirm "auslegt" (layout). Zu den wichtigeren Layouts zählen die Folgenden:

- **BorderLayout:** hier gibt es fünf Regionen und Widgets müssen explizit einer Region zugewiesen werden
- **FlowLayout:** ist das einfachste Layout, Widgets werden einfach von links nach rechts nebeneinander ausgelegt
- **GridLayout:** der verfügbare Platz wird in gleich große Flächen aufgeteilt, z.B. 3 mal 2



Sehen wir uns ein paar Beispiele dazu an. Um den BorderLayout zu verwenden würden wir folgenden Code verwenden:

```
setLayout(new BorderLayout());
add(new JButton("EAST"), EAST);
add(new JButton("WEST"), WEST);
...
```

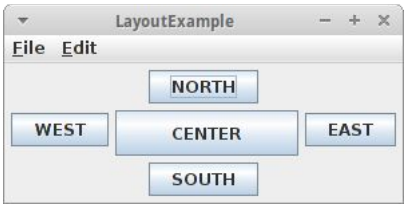
Für den FlowLayout sieht das dann so aus:

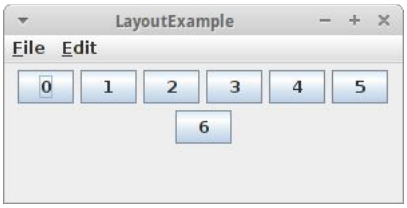
```
setLayout(new FlowLayout());
add(new JButton("0"));
add(new Jbutton("1"));
...
```

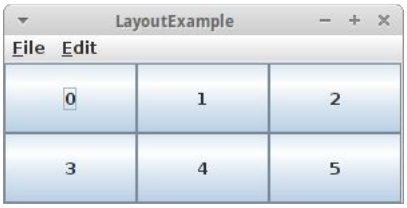
und für den GridLayout so:

```
setLayout(new GridLayout(2, 3));
add(new JButton("0"));
add(new Jbutton("1"));
...
```

Grafisch sieht das Ganze dann so aus:







JPanel

Was die GCompound für GraphicsProgramme war ist das JPanel für Swing: es erlaubt es uns mehrere Widgets zu einem neuen Widget zusammenzufassen. Details zur JPanel Klasse sehen wir weiter unten im Projekt "Quiz".

Review

Glückwunsch! Wie wir gleich sehen werden, haben wir jetzt das Rüstzeug um fast beliebige grafische Benutzeroberflächen (UIs) zu basteln. Wir wissen jetzt wie man mit

- Labels
- Buttons
- Textfields
- Checkboxes
- Radiobuttons
- und Comboboxes

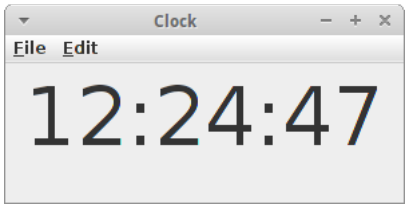
arbeitet. Außerdem haben wir etwas über verschiedene Layouts erfahren, und kurz die Bekanntschaft des JPanel gemacht.

Mindestens genauso wichtig war aber die Vertiefung zu Instanzvariablen. Es sollte jetzt etwas klarer sein, was diese von lokalen Variablen unterscheidet, und wann man welche der beiden verwendet.

Projekte

In den Projekten in diesem Kapitel werden wir viele UIs erstellen. Manche brauchen wir später noch, um sie mit Leben zu füllen. Es gibt wieder viel zu tun.

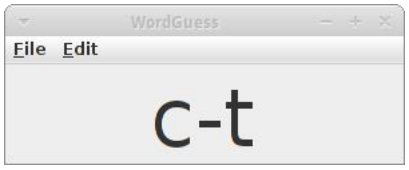
Clock



Wir wollen eine kleine Uhr schreiben. Dazu verwenden wir die *Date* Klasse und deren Methoden *getHours()*, *getMinutes()* und *getSeconds()*. Für die Anzeige selbst verwenden wir einen JLabel. Und natürlich sollte der Text des JLabels sich einmal pro Sekunde (besser zweimal) verändern. Dazu kann man die *setText()* Methode des JLabels verwenden.

Unter Umständen macht es Sinn eine Methode *padWithZeros()* zu schreiben, die sicher stellt, dass anstelle von "6" Minuten "06" Minuten angezeigt werden.

WordGuess



WordGuess ist sozusagen eine grafische Version von Hangman aus dem letzten Kapitel. Es geht darum ein Wort zu erraten indem man Buchstaben eingibt.

Man beginnt damit, dass man ein zufälliges Wort mit der Methode *pickRandomWord()* auswählt. Dieses Wort sollte man als Instanzvariable *wordToGuess* speichern. Dann sollte man daraus ein Wort mit lauter "-" Strichen machen, *wordShown*, auch eine Instanzvariable. Dann kreiert man einen JLabel und fügt diesen im Norden dazu:

```
wordLbl = new JLabel(wordShown);
add(wordLbl, NORTH);
wordLbl.addKeyListener(this);

wordLbl.requestFocus();
```

Interessant ist die Art und Weise wie wir hier den KeyListener hinzufügen: zunächst einmal ist er Teil des Labels, und zum zweiten übergeben wir "this" als Parameter. Momentan brauchen wir das noch nicht zu verstehen, aber effektiv führt es dazu, dass wir auf KeyEvents hören können. Die *requestFocus()* Methode ist nötig, damit der KeyListener auch funktioniert.

Was bleibt ist die Methode *keyTyped(KeyEvent e)* zu implementieren. Diese wird aufgerufen wenn der Nutzer eine Taste drückt, was dann passieren soll ist ähnlich wie bei Hangman, wir prüfen ob der Buchstabe im *wordToGuess* vorhanden ist, und updaten den Label falls er ist. Natürlich macht es auch noch Sinn zu zählen wie viel Versuche benötigt wurden um das Wort zu erraten.

StopWatch

Eine Stoppuhr muss eine höhere Genauigkeit haben als die Klasse *Date* liefert. Dafür gibt es die Systemmethode

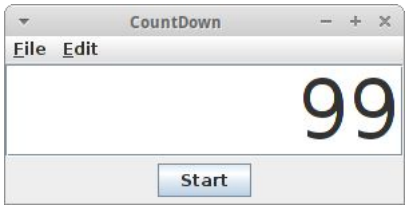
```
long time = System.currentTimeMillis();
```

welche uns die Millisekunden liefert, die seit 0 Uhr des 1.1.1970 verstrichen sind. Wenn wir eine Zeit die in Millisekunden gegeben ist durch 1000 teilen erhalten wir die Sekunden, wenn wir Modulo 1000 nehmen erhalten wir die Millisekunden.

Die Anzeige soll animiert sein, deswegen macht es Sinn das Ganze in einen GameLoop zu stecken. Außerdem macht es Sinn einen Delay von 20ms zu haben, sonst kommt die Anzeige nicht nach mit dem Anzeigen. Und wir sollten zwei Knöpfe mit einbauen, einen zum Starten und einen zum Pausieren.

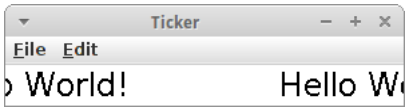
Den primitiven Datentyp *long* haben wir bisher noch nicht gesehen. Er verhält sich wie ein *int*, ist also für Ganzzahlen gedacht. Der einzige Unterschied, während ints 32 Bit sind, sind die longs 64 Bit, d.h. sie eignen sich für größere Zahlen (32-bit Zahlen sind zwischen -2^31 und +2^31 und 64-bit Zahlen sind zwischen -2^63 und +2^63).

CountDown



Ganz ähnlich wie die Stoppuhr funktioniert der CountDown. Anstelle eines JLabels verwenden wir aber ein JTextField. Der Vorteil ist, dass man dieses editieren kann, d.h. man kann die Zahl einstellen von der man aus rückwärts zählen möchte. Sobald der Nutzer dann auf den "Start" Knopf drückt, soll der Countdown beginnen.

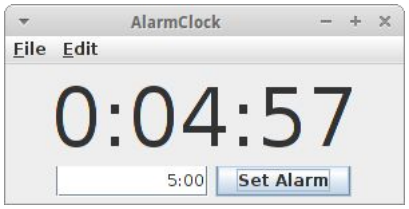
Ticker



Eine Laufschrift (Werbe-Ticker) ist relativ einfach zu implementieren. Wir verwenden dazu einen JLabel. In einem GameLoop bewegen wir in dann alle 50 Millisekunden um 5 Pixel nach links mittels

```
lbl.move(-5, 0);
```

AlarmClock



Eine Alarmuhr zu schreiben, die mit einem Stunden:Minuten:Sekunden Format arbeitet ist überraschen kompliziert. Allerdings wenn man die Methoden für die Konvertierung vom Stunden:Minuten:Sekunden nach Sekunden (*convertTimeInSeconds()*) und von Sekunden nach Stunden:Minuten:Sekunden (*convertSecondsInTime()*) schon hat, dann ist es gar nicht so schwer, und eigentlich identisch mit dem CountDown Projekt.

Für das Programm verwenden wir einen großen JLabel, den wir im Norden platzieren. Außerdem gibt es ein JTextField im Süden für die Eingabe der Alarmzeit im Stunden:Minuten:Sekunden Format. Und es gibt einen JButton über den der Alarm gestartet wird.

Es macht Sinn zwei Instanzvariablen zu verwenden:

```
private long alarmTime = -1;
private boolean alarmStarted = false;
```

Die erste ist einfach die Zeit in Sekunden, und die zweite wird verwendet um dem GameLoop mitzuteilen, dass etwas zu tun ist:

```
while (true) {
    if (alarmStarted) {
        // display remaining time
        ...
    }
    pause(DELAY);
}
```

Wenn der JButton gedrückt wird, dann wir zum einen alarmTime auf die Sekunden gesetzt, und alarmStarted wird auf *true* gesetzt, damit der GameLoop weiß, dass er jetzt was anzeigen soll.

Editor

Als nächstes steht ein Texteditor auf unserem Plan. Dieser besteht zum Einen aus einem JTextField für den Dateinamen und zwei JButtons, einer zum Laden, der andere zum Speichern von Dateien. Diese drei Widgets platzieren wir im Norden, also dem unteren Teil.

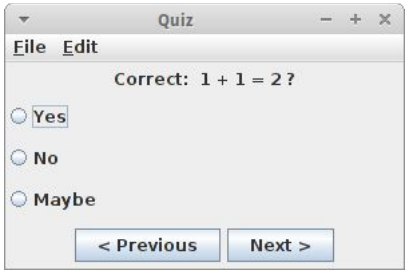
In den Mittelteil, CENTER, kommt eine JTextArea:

```
display = new JTextArea();
display.setFont(
    new Font("Courier", Font.BOLD, 18)
);
add(display, CENTER);
```

Eine JTextArea ist wie ein JTextField, nur dass man da auch Mehrzeilentext eingeben kann.

Wie das Laden und Speichern von Dateien geht, lernen wir im nächsten Kapitel.

Quiz



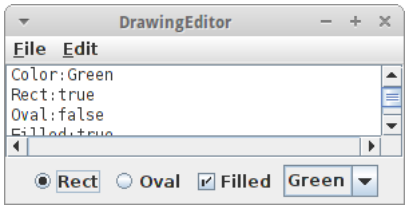
Wir wollen eine UI für ein MultipleChoice Quiz schreiben. Diese besteht aus einer Frage, also einen JLabel, den wir im oberen Bereich (NORTH) platzieren. Dann folgen die möglichen Antworten. Dies sind natürlich Checkboxes. Da sie zusammen gehören, gruppieren wir sie in eine Buttongroup. Die Checkboxes kommen in den mittleren Bereich (CENTER). Schließlich, wollen wir auch noch zwei Navigations Knöpfe hinzufügen, im unteren Bereich. Das Programm soll keine weitere Funktion haben, die kommt im übernächsten Kapitel.

Dieses Programm ist ein schönes Beispiel wie man JPanels einsetzen kann. Multiplechoicefragen bestehen nicht immer aus drei Antworten. Manchmal sind es weniger, manchmal mehr. Also macht es Sinn, die Fragen zusammenzufassen und in ein JPanel einzufügen.

```
JPanel answersPnl = new JPanel();
answersPnl.setLayout(new GridLayout(3, 1));
JRadioButton btn1 = new JRadioButton(answer1);
answersPnl.add(btn1);
...
add(answersPnl, CENTER);
```

Dieses JPanel fügen wir dann in den mittleren Bereich.

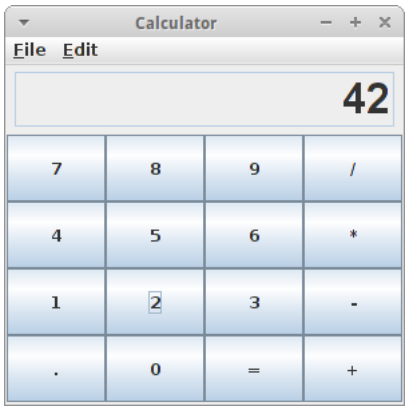
DrawingEditor



Im übernächsten Kapitel wollen wir einen DrawingEditor schreiben. Bis dahin können wir schon einmal etwas Vorarbeit leisten. Die Idee ist, dass wir zwischen den Formen Rechteck und Kreis mittels zweier Radiobuttons auswählen können. Zusätzlich wollen wir einstellen können ob die Formen ausgefüllt sind oder nicht, das geht am besten mit einer Checkbox. Und schließlich, möchten wir noch die Farbe der Formen bestimmen können.

Challenges

Calculator



Unser nächstes Projekt ist ein kleiner Taschenrechner. Dieser besteht aus einem JTextField (*display*) und 16 JButtons. Am besten wir platzieren das JTextField im Norden, und die JButtons in ein 4x4 GridLayout.

Bei der Programmierung der Logik des Rechners müssen wir etwas überlegen. Zunächst macht es Sinn zwei Instanzvariablen einzuführen:

```
private double operand1 = 0;
private char operation = '+';
```

Wenn wir z.B. "6 - 2" mit einem Taschenrechner ausrechnen, dann geben wir ja zuerst die Zahl "6" ein, dann das "-" und danach die "2". Wir müssen uns also zwischendurch sowohl die "6" als auch das "-" merken, deswegen die beiden Instanzvariablen.

In der *actionPerformed()* Methode müssen wir also unterscheiden zwischen dem "=" Zeichen, den Operatoren "+", "-","*" und "/", und den Ziffern.

Wenn Ziffern eingegeben werden, dann fügen wir die einfach zum *display* hinzu:

```
char cmd = e.getActionCommand().charAt(0);
display.setText(display.getText() + cmd);
```

Wenn ein Operator gedrückt wurde, dann müssen wir die Instanzvariablen *operand1* und *operation* setzen, also

```
operand1 = Double.parseDouble(display.getText());
display.setText("");
operation = cmd;
```

Und wenn das "=" Zeichen gedrückt wurde, dann müssen wir die Berechnung durchführen und anzeigen:

```
double operand2 = Double.parseDouble(display.getText());
double result = calculate(operand1, operand2, operation);
display.setText("" + result);
```

Es bleibt also lediglich die Methode *calculate(double operand1, double operand2, char operation)* zu implementieren.

Fragen

1. Es gibt lokale Variablen, Instanzvariablen und Konstanten. Erklären Sie den Unterschied.

2. Was muß man ändern, damit aus 'PI' eine Konstante wird?
   `double PI = 3.14;`

3. Im folgenden Beispiel gibt es mehrere Variablen, teilweise mit gleichem Namen. Beschreiben Sie wie die Variablen zusammenhängen, und welche wo gültig sind.

```
public class Lifetime {

    public void run() {
      int i = 3;
      cow(i);
    }

    private void cow( int n ) {
      for (int i=0; i<3; i++) {
          ...
      }
    }
  }
```

4. Nennen Sie drei verschiedene LayoutManager.

5. Skizzieren Sie wie die UI für den folgenden Code aussehen würde.

6. Wenn Sie auf einen JButton klicken, welche Art von Ereignis (Event) wird dann ausgelöst?

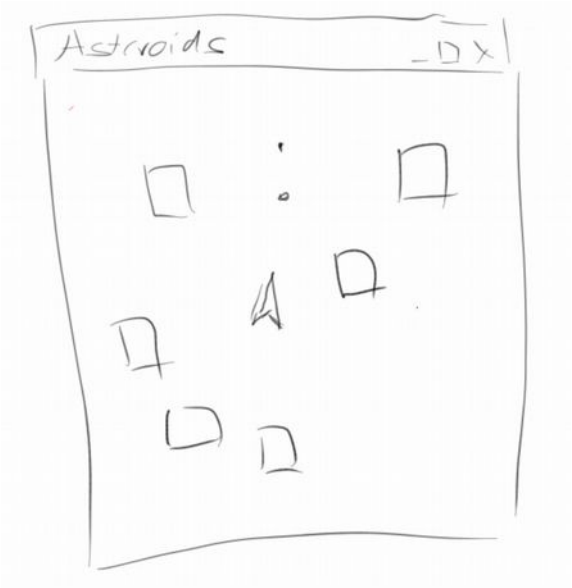
Referenzen

In diesem Kapitel ist die bevorzugte Referenz das Buch von Eric Robert [1]. Ein schönes, allerdings anspruchvolles Tutorial ist das von Oracle, den Machern von Java [2].

[1] The Art and Science of Java, von Eric Roberts, Addison-Wesley, 2008

[2] The Swing Tutorial, docs.oracle.com/javase/tutorial/uiswing/

Asteroids



In diesem Kapitel gibt es wieder viele Spiele. Aber vorher müssen wir uns mit Objekt-Orientierung beschäftigen. Wir lernen die Grundpfeiler der Objekt-Orientierung kennen, nämlich Vererbung und Komposition. Wir werden auch Arrays kennenlernen, und wie man mit mehrdimensionalen Arrays Bilder manipuliert. Außerdem lernen wir wie man mit Tastaturereignissen (KeyEvents) arbeitet und die Klasse GCompound wird kurz vorgestellt. Wir beginnen mit den Arrays.

Arrays

Was sind Arrays? Ein Eierkarton ist ein Array. Und zwar ist es ein Array für zehn Eier. D.h. aber nicht dass da auch zehn Eier drin sind, manchmal sind nur drei Eier drin.

Offensichtlich sind also Arrays ganz praktisch und deswegen betrachten wir Arrays in Java. Nehmen wir einmal an wir wollten ein Array für zwölf Ganzzahlen anlegen, dann würden wir schreiben:

```
int[] eggs;
eggs = new int[10];
```

In der ersten Zeile deklarieren wir ein Array. Wir sagen dass es sich bei der Variablen *eggs* um ein Array vom Typ *int* handelt, indem wir hinter dem Datentyp einfach eckige Klammern schreiben. In der zweiten Zeile legen wir dann das Array an und sagen, dass es Platz für zehn *int*'s geben soll.

Wir können Arrays von beliebigen Datentypen anlegen, z.B. könnten wir auch ein Array mit vier GOvals anlegen:

```
GOval[] circles = new GOval[4];
```

Alle Arrays haben zwei wichtige Eigenschaften:

1. sie sind immer vom selben Datentyp, man sagt auch sie sind homogen, und
2. sie sind geordnet, d.h., sie sind durchnummeriert beginnend mit 0.

Mit Arrays arbeiten

Nachdem wir ein Array deklariert und angelegt haben, müssen wir es mit Werten füllen. Das können wir von Hand machen:

```
eggs[0] = 0;
eggs[1] = 2;
eggs[2] = 4;
...
eggs[9] = 18;
```

Wir weisen also dem ersten Element im Array (dem Element Nummer 0) den Wert 0 zu, dem zweiten Element den Wert 2 usw. Wir können die Zuweisung aber auch mit einer Schleife machen:

```
for (int i=0; i<10; i++) {
    eggs[i] = readInt("?");
}
```

oder mit dem folgende Trick (der allerdings nur beim Anlegen funktioniert):

```
int[] eggs = { 2, 4, 6, 8 };
```

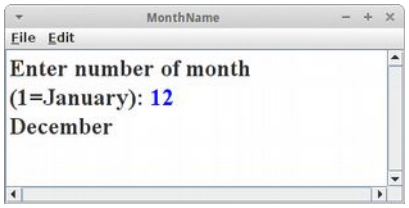
Wenn wir auf ein Element zugreifen wollen, müssen wir seine Hausnummer angeben. Also auf das dritte Element greifen wir mit :

```
println( eggs[2] );
```

zu. Wenn wir alle Elemente ausgeben wollen, geht das am besten mit einer Schleife:

```
for (int i=0; i<eggs.length; i++) {
    println( eggs[i] );
}
```

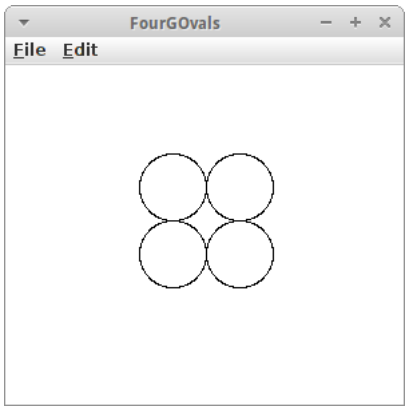
Übung: MonthName



Ein nützliches Beispiel ist die Konvertierung vom Monat als Zahl, also z.B. 12, in den Monatsnamen, also z.B. Dezember. Man könnte das mit einer langen *if* oder *switch* Bedingung machen, aber man kann das auch sehr elegant mit Arrays lösen:

```
public class MonthName extends ConsoleProgram {

    private String[] monthName = { "January", "February", "March", "April",
            "May", "June", "July", "August", "September", "October",
            "November", "December" };

    public void run() {
        int monthNr = readInt("Enter number of month (1=January): ");
        println(monthName[monthNr - 1]);
    }
}
```

Array von Objekten



Im Prinzip sind Arrays nicht weiter kompliziert. Allerdings gibt es manchmal eine kleine Verwirrung wenn man Arrays von Objekten hat. Schauen wir uns dazu zunächst die Deklaration eines Arrays von GOvals an:

```
GOval[] circles = new GOval[4];
```

und vergleichen es mit dem Anlegen eines neuen Kreises:

```
GOval circle = new GOval(100,100,50,50);
```

Was ist der Unterschied? Im ersten Fall legen wir Platz für vier GOvals an. Wir legen aber noch keine Kreise selbst an. Im zweiten Fall dagegen legen wir einen Kreis an. Wenn wir also ein Array mit vier Kreisen anlegen wollen (und nicht nur Platz für vier Kreise), dann müssen wir ein bisschen mehr Code schreiben:

```
GOval[] circles = new GOval[4];
circles[0] = new GOval(100, 66, 50, 50);
circles[1] = new GOval(100, 116, 50, 50);
circles[2] = new GOval(150, 66, 50, 50);
circles[3] = new GOval(150, 116, 50, 50);
```

Mehrdimensionale Arrays

Eindimensionale Arrays sind ganz lustig und sparen uns viel Schreibarbeit. Aber wirklich cool sind zweidimensionale Arrays. Wir fangen ganz einfach mit einem Schachspiel an.

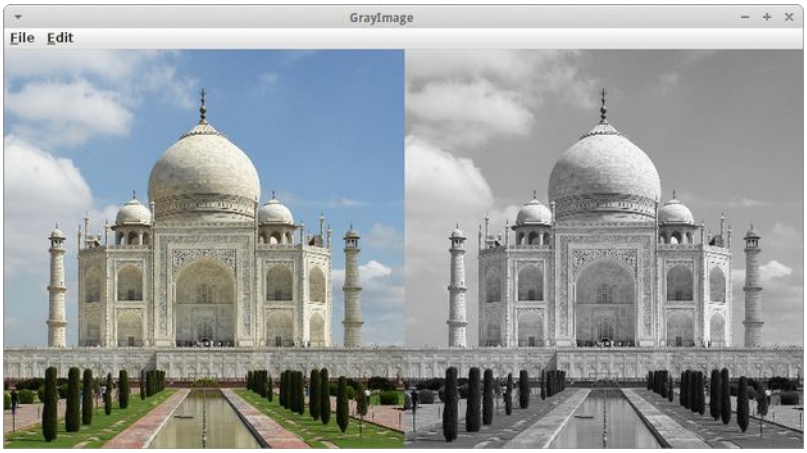
```
private char[][] chess = {
    { 'r', 'n', 'b', 'q', 'k', 'b', 'n', 'r' },
    { 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p' },
    { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
    { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
    { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
    { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' },
    { 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P' },
    { 'R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R' } };
```

Es handelt sich hier also um ein 8 mal 8 Array von chars. Kleinbuchstaben stehen für schwarz, und Großbuchstaben für weiß. Wenn wir das Spielfeld ausgeben wollen, dann könnten wir das mit zwei verschachtelten *for* Schleifen tun:

```
private void printChessBoard() {
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            print(chess[i][j]);
        }
        println();
    }
}
```

Übung: GrayImage

Bilder sind auch zweidimensionale Arrays. Als kleine Übung wollen wir ein Farbbild in ein Graubild umwandeln. Zunächst laden wir das Bild mittels der GImage Klasse:



Taj Mahal, Bildquelle Wikipedia [1]

```
GImage image = new GImage("Taj_Mahal_(Edited).jpeg");
```

Als nächstes müssen wir an die Pixel drankommen. Das geht mit der Methode *getPixelArray()* der Klasse GImage:

```
int[][] array = image.getPixelArray();
int height = array.length;
int width = array[0].length;
```

Diese liefert uns ein zweidimensionales Array von *int*s. Jeder dieser *int* entspricht einem Pixel. Wenn wir also den Pixel an Position x=5 und y=22 möchten, dann geht das mittels

```
int pixel = array[5][22];
```

Jeder dieser Pixel enthält dessen rot, grün und blau Werte, und mittels

```
int r = GImage.getRed(pixel);
int g = Gimage.getGreen(pixel);
int b = GImage.getBlue(pixel);
```

erhalten wir diese. Um daraus ein Graubild zu erstellen, verwenden wir die Formel die auch Gimp verwendet [2]:

```
lum = 0.21 * r + 0.72 * g + 0.07 * b;
```

das packen wir dann wieder in das zweidimensionales Array

```
array[5][22] = GImage.createRGBPixel(lum, lum, lum);
```

und am Ende machen wir daraus ein neues GImage

```
GImage grayImage = new GImage(array);
```

Objekt-Orientierung

Im zweiten Teil dieses Kapitels wollen wir unsere Kenntnisse bzgl der Objektorientierung vertiefen. Die zwei großen Themen die anstehen sind zum einen *Vererbung* ("is a" Beziehung) und zum anderen *Komposition* ("has a" Beziehung). Wir beginnen mit einem kleinen Spiel, dem *MarsLander*.

Übung: MarsLander

Elon Musk will ja im kommenden Jahrzehnt die ersten Menschen auf den Mars schicken. Karel hat sich freiwillig gemeldet, und muss jetzt erst mal die Landung üben. Dazu gibt es einen Simulator den *MarsLander*. Es geht darum ein Raumschiff sicher auf dem Mars zu landen. Dazu können wir mit den Pfeiltasten (nach oben und nach unten) das Raumschiff abbremsen oder beschleunigen. Wenn unser Geschwindigkeit beim Touchdown zu hoch ist, sterben wir.

Der Top-Down Ansatz bietet sich an: Betrachten wir die *run()* Methode:

```
public void run() {
    setup();
    waitForClick();

    // game loop
    while (spaceShip != null) {
        moveSpaceShip();
        checkForCollision();
        pause(DELAY);
    }
    displayGameOver();
}
```

Wie üblich fangen wir mit dem *setup()* an. Nach dem Setup warten wir bis der Nutzer mit der Maus einmal auf den Bildschirm klickt um das Spiel zu starten. Danach beginnt der GameLoop und der sieht genauso aus wie bei unserer letzten Animation, dem Billard. Interessant ist jetzt, dass wir keine Endlosschleife mehr haben, sondern eine Schleife mit Abbruchkriterium: nämlich wenn es kein SpaceShip mehr gibt, also *spaceShip == null*, dann soll das Spiel aufhören.

Es gibt drei Instanzvariablen,

```
private GPolygon spaceShip;
private int vy = 0;
private int vx = 0;
```

also das spaceShip, sowie dessen Geschwindigkeiten vx und vy.

In der setup() Methode wird das spaceShip initialisiert, der Code ist identisch mit dem der Übung aus Kapitel zwei,

```
private void setup() {
    spaceShip = new GPolygon();
    spaceShip.addVertex(0, -SPACE_SHIP_SIZE);
    spaceShip.addVertex(-2 * SPACE_SHIP_SIZE / 3, SPACE_SHIP_SIZE);
    spaceShip.addVertex(0, SPACE_SHIP_SIZE / 2);
    spaceShip.addVertex(2 * SPACE_SHIP_SIZE / 3, SPACE_SHIP_SIZE);
    add(spaceShip, (getWidth()-SPACE_SHIP_SIZE) / 2, SPACE_SHIP_SIZE);

    addKeyListeners();
}
```

und wir fügen einen *KeyListener* hinzu. Wir wollen das spaceShip ja mittels der Tastatur (keyboard) steuern, und deswegen müssen wir auf Tastenereignisse (KeyEvents) hören. Das ist also vollkommen analog zu den MouseEvents und dem MouseListener.

Die *moveSpaceShip()* Methode ist absolut trivial:

```
private void moveSpaceShip() {
    vy += GRAVITY;
    spaceShip.move(vx, vy);
}
```

Da wir uns im Schwerefeld (GRAVITY) des Mars befinden, erhöht sich unsere Geschwindigkeit in jedem Schritt. Und in jedem Schritt bewegen wir das spaceShip um den Betrag der Geschwindigkeit.

In der *checkForCollision()* Methode checken wir, ob wir schon auf der Marsoberfläche (also unten) angekommen sind:

```
private void checkForCollision() {
    double y = spaceShip.getY();
    if (y > (getHeight() - SPACE_SHIP_SIZE)) {
        spaceShip = null;
    }
}
```

Falls ja, dann setzen wir das spaceShip einfach auf *null*. "null" heißt soviel wie "nicht initialisiert" oder "existiert nicht" oder "gibt es nicht". Das ist ein vordefinierter Wert, den alle Objekte haben bevor sie mittels *new* erzeugt werden. Wir können aber auch Objekte explizit auf *null* setzen, und das bedeutet das wir das Objekt löschen. In unserem Beispiel verwenden wir das um die Endlosschleife zu beenden.

Was noch bleibt sind die Tastenereignisse. Ähnlich wie es bei der Maus die *mousePressed()* Methode gibt, gibt es auch eine *keyPressed()* Methode:

```
public void keyPressed(KeyEvent e) {
    switch (e.getKeyCode()) {
    case 38: // up
        vy--;
        break;
    case 40: // down
        vy++;
        break;
    }
}
```

Wir wollen natürlich wissen welche Taste gedrückt wurde und das erfahren wir von der *getKeyCode()* Methode de KeyEvents. Jede Taste hat ihren eigenen KeyCode, und für die Pfeil-Oben Taste ist das die 38 und für die Pfeil-Unten Taste ist das die 40.

So, jetzt können wir spielen, bzw. trainieren.

Vererbung

Was hat der *MarsLander* mit Vererbung zu tun? Noch nicht viel. Aber schauen wir uns den Code an. Was uns ein bisschen stören sollte sind die drei Zeilen,

```
private GPolygon spaceShip;
private int vy = 0;
private int vx = 0;
```

denn bei *vx* und *vy* handelt es sich um die Geschwindigkeit des spaceShip, die gehören also eigentlich zum spaceShip. Nehmen wir an wir hätten mehrere spaceShips, oder wir hätten ganz viele Asteroiden die sich durch die Gegend bewegen, dann hätten wir ganz viele vx's und vy's. Und das wird total unübersichtlich und hässlich.

Um das zu verhindern machen wir folgendes: wir deklarieren eine neue Klasse namens GSpaceShip und alles was mit spaceShip zu tun hat packen wir in diese Klasse:

```
public class GSpaceShip extends GPolygon {
    // constants
    private final int GRAVITY = 1;
    // instance variables
    public int vy = 0;
    public int vx = 0;

    public GSpaceShip(int SPACE_SHIP_SIZE) {
        super();

        addVertex(0, -SPACE_SHIP_SIZE);
        addVertex(-2 * SPACE_SHIP_SIZE / 3, SPACE_SHIP_SIZE);
        addVertex(0, SPACE_SHIP_SIZE / 2);
        addVertex(2 * SPACE_SHIP_SIZE / 3, SPACE_SHIP_SIZE);
    }

    public void move() {
        vy += GRAVITY;
        move(vx, vy);
    }
}
```

Als erstes sehen wir, dass es sich bei *GSpaceShip* um ein *GPolygon* handelt, denn es sagt "GSpaceShip extends GPolygon", also GSpaceShip *ist ein* GPolygon. GSpaceShip erbt also alle Eigenschaften und Methoden von GPolygon. Deswegen sagt man auch *Vererbung* ist eine "is a" Beziehung.

Als zweites sehen wir, dass die Instanzvariablen *vx* und *vy* jetzt Instanzvariablen des spaceShips sind, sie sind also da wo sie hingehören.

Als drittes schauen wir uns den Konstruktor an: dort sehen wir in der ersten Zeile ein "*super()*". Die Methode super() tut nichts anderes als den Konstruktor der Superklasse aufzurufen, also der Elternklasse. In unserem Fall ist das GPolygon(). Danach sehen wir, wie wir uns selbst (wir sind ja jetzt ein GPolygon) Vertices hinzufügen. D.h. im Konstruktor bestimmen wir unser Aussehen.

Als letztes sehen wir, dass wir eine neue Methode namens *move()* hinzugefügt haben. Da GSpaceShip jetzt ja seine eigene Geschwindigkeit kennt, kann es sich ja auch selbst bewegen.

Vererbung hat also viele Vorzüge: vor allem führt sie dazu, dass Klassen selbstständiger werden, und weniger Abhängigkeiten haben. Man sagt auch die Klasse übernimmt Verantwortung über ihre eigenen Attribute (Variablen) und Verhalten (Methoden). Diese geringeren Abhängigkeiten führen auch zu einer geringeren Kopplung, die dazu führt, dass unser Code weniger kompliziert wird.

Schauen wir uns die Vereinfachungen im *MarsLander2* an. Zunächst brauchen wir nur noch eine Instanzvariable:

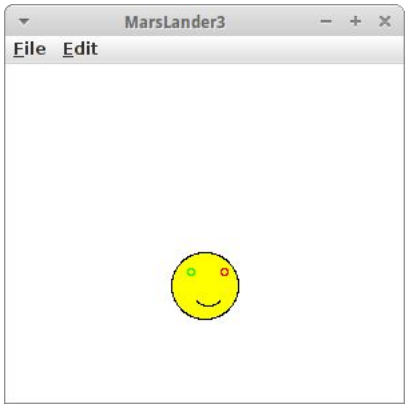
```
    private GSpaceShip spaceShip;
```

und außerdem werden die *setup()* und *moveSpaceShip()* Methoden viel kürzer:

```
    private void setup() {
        spaceShip = new GSpaceShip(SPACE_SHIP_SIZE);
        add(spaceShip, (getWidth()-SPACE_SHIP_SIZE) / 2, SPACE_SHIP_SIZE);
        addKeyListeners();
    }

    private void moveSpaceShip() {
        spaceShip.move();
    }
```

Das ist schon ziemlich cool. Den wahren Wert dieser Vereinfachungen werden wir aber erst schätzen lernen wenn es daran geht *Asteroids* zu programmieren.

Komposition



Das zweite wichtige Konzept der Objektorientierung ist die Komposition. Wie wir gesehen haben, kann man neue Klassen (GSpaceShip) durch Vererbung von einer existierenden Klasse (GPolygon) erzeugen. Man kann aber auch neue Klassen erzeugen, indem man sie aus mehreren existierenden Klassen zusammensetzt, also komposiert.

Als Beispiel schreiben wir eine Klasse GSmiley.

```
public class GSmiley {

    public GSmiley(int SIZE) {
        GOval face = new GOval(SIZE, SIZE);
        face.setFilled(true);
        face.setFillColor(Color.YELLOW);
        add(face);

        GOval leftEye = new GOval(SIZE/10, SIZE/10);
        leftEye.setColor(Color.GREEN);
        add(leftEye, SIZE/4, SIZE/4);
        GOval rightEye = new GOval(SIZE/10, SIZE/10);
        rightEye.setColor(Color.RED);
        add(rightEye, 3*SIZE/4, SIZE/4);

        GArc mouth = new GArc(SIZE/2, SIZE/2, 225, 90);
        add(mouth, 0.3*SIZE, 0.3*SIZE);
    }
}
```

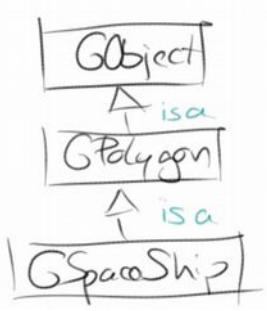
Unser GSmiley besteht aus verschiedenen Komponenten, es hat also ein *face*, ein *leftEye*, ein *rightEye* und ein *mouth*. Im Konstruktor basteln wir also ein neues Objekt aus mehreren alten. Das ist Komposition. Deswegen sagt man auch dass *Komposition* eine "has a" Beziehung ist, denn GSmiley hat ein *face*, ein *leftEye*, ein *rightEye* und ein *mouth*.

GCompound

Man kann Vererbung und Komposition auch mischen. Wenn wir beim GSmiley Beispiel noch "extends GCompound" zur Klassendeklaration hinzufügen, dann können wir GSmiley auch in unserem MarsLander verwenden. Wir müssen dann in der ersten Version des MarsLanders einfach "GPolygon" durch "GSmiley" ersetzen.

Vererbung vs Komposition

Wann verwendet man Vererbung und wann Komposition? Eine Daumenregel lautet, wenn möglich sollte man Komposition verwenden. Das hat damit zu tun, dass es in Java keine Mehrfachvererbung gibt. Also eine Klasse kann keine zwei Eltern haben. Diese Einschränkung gibt es bei Komposition nicht, im Prinzip kann eine Klasse aus beliebig vielen Komponenten bestehen.



Eine kleine Anmerkung noch: mit Mehrfachvererbung meinen wir immer eine Klasse hat mehrere Elternklassen. Das ist nicht erlaubt. Es ist aber durchaus möglich dass eine Klasse eine Elternklasse hat, und diese Elternklasse hat wieder eine Elternklasse, also sozusagen die Großelternklasse der ursprünglichen. Also z.B. GObject ist die Großelternklasse der Klasse GSpaceShip. Das ist erlaubt.

Review

Mit den Prinzipien Vererbung und Komposition haben wir den Kern der Objektorientierung erreicht und geknackt. Wir haben gelernt wie man einer existierenden Klasse mittel Vererbung zusätzlich Eigenschaften geben kann. So kann sich ein GPolygon nicht selbstständig bewegen, da es keine Geschwindigkeit hat. Die Klasse GSpaceShip, die ja eigentlich auch ein GPolygon ist, kennt aber seine eigene Geschwindigkeit, und kann sich selbstständig bewegen. Weiter haben wir gesehen, dass man mittels Komposition aus mehreren Klassen eine neue Klasse zusammenbauen kann. Beides ist sehr nützlich, wie wir sehen werden.

Zusätzlich haben wir noch ein paar andere nützliche Dinge, wie z.B.

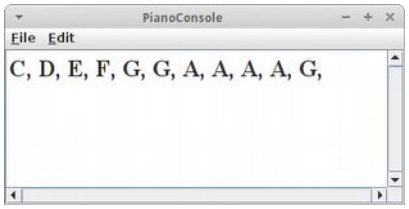
- Arrays
- mehrdimensionale Arrays
- Bildverarbeitung
- Tastaturereignisse
- und die Klasse GCompound

kennengelernt.

Projekte

Die Projekte in diesem Kapitel fangen an richtig Spaß zu machen. Los geht's.

PianoConsole



Als erste Anwendung für Arrays schreiben ein kleines Musikprogramm. In unserem Array speichern wir die Melodie:

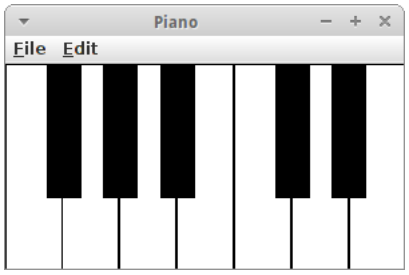
```
      private String[] tune = { "C", "D", "E", "F", "G",
  "G" };
```

Zum einfachen Abspielen von Audio-Dateien können wir die *AudioClip* Klasse verwenden:

```
AudioClip audioClip = getAudioClip(getCodeBase(), "sound.wav");
audioClip.play();
pause(500);

audioClip.stop();
```

Der *AudioClip* Klasse sagt man welche Datei sie abspielen soll, mit der Methode *play()* wird dann das Abspielen gestartet und mit *stop()* beendet. Wenn man jetzt für die verschiedenen Noten verschiedene wav-Dateien hat, also z.B. "C.wav", "D.wav", usw. dann kann man so Melodien abspielen, indem man mit einer Schleife die einzelnen Noten in dem *tune* Array durch iteriert.

Piano



Konsolenanwendungen sind immer etwas langweiliger, und ehrlich wer würde dafür schon Geld ausgeben? Wir haben aber ja schon im zweiten Semester eine UI für unser Piano geschrieben. Natürlich würde wir unser Klavier über die Maus steuern, also MouseListener im *setup()* hinzufügen.

Die Frage die sich allerdings stellt, wie wissen wir welche Taste gedrückt wurde? Interessanterweise können wir dafür unsere *getElementAt()* Methode verwenden:

```
public void mouseClicked(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    GObject obj = getElementAt(x, y);
    if (obj != null) {
        ...
    }
}
```

Darüber können wir also herausfinden, auf welches GRect gedrückt wurde. Jetzt gibt es drei Alternativen weiter zu machen:

1. Wenn das GRect einen Namen hätte wäre die Welt ganz einfach. Wir können das mit Vererbung erreichen: wir definieren eine neue Klasse *GKey*, die ein GRect ist und noch zusätzlich ein Attribute für den Namen hat.
2. Wir merken uns irgendwo die x-Koordinate der Tasten. Mit *obj.getX()* können wir diese ja erhalten, und viola wissen wir welche Taste gedrückt wurde.
3. Oder wir halten uns Referenzen zu allen Tasten in einem Array.

Diese dritte Möglichkeit wollen wir kurz etwas näher betrachten. Dazu brauchen wir ein Array als Instanzvariable

```
private GRect[] keys = new GRect[12];
```

Wenn wir die Tasten erzeugen, dann speichern wir die einfach in unserem Instanzarray:

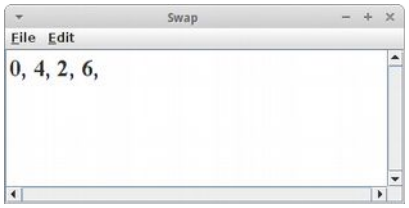
```
int keyCounter = 0;
// draw 8 white keys
for (int i = 0; i < 7; i++) {
    keys[keyCounter] = new GRect(WIDTH / 7, HEIGHT - HEIGHT_OFFSET);
    add(keys[keyCounter], i * WIDTH / 7, 0);
    keyCounter++;
}
```

Und jetzt können wir in unserer *mouseClicked()* Methode einfach auf Gleichheit testen:

```
for (int i = 0; i < keys.length; i++) {
    if (obj == keys[i]) {
        AudioClip audioClip = getAudioClip(getCodeBase(), "music/"
            + tunes[i] + "4.wav");
        println(tunes[i] + "4.wav");
        audioClip.play();
    }
}
```

Wenn wir jetzt die App auch noch auf dem Handy zum Laufen kriegen würden, dann wären wir reich! (Nächstes Jahr...)

Swap



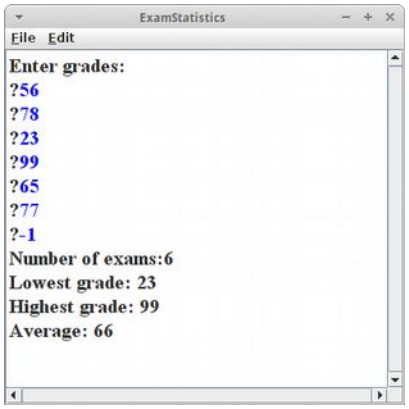
In diesem Projekt wollen wir zwei Elemente eines Arrays vertauschen. In dem Array

```
int[] arr = { 0, 2, 4, 6 };
```

möchten wir das Element an der zweiten Position (also die "2") mit dem Element an der dritten Position (also der "3") vertauschen. Das wollen wir mit einer Methode *swap(int[] arr)* machen, die ein Array als Übergabeparameter hat.

Zwei Dinge wollen wir in dieser Übung lernen: Erstens in Arrays beginnen wir immer mit 0 zu zählen, und Arrays werden als Referenz übergeben, d.h., wenn wir ein Array als Übergabeparameter an eine Methode übergeben, dann wir dieses im Original übergeben. Alle Änderungen die wir in der Methode daran vornehmen sind permanent, also ändern das Array.

ExamStatistics



Als weiteres Beispiel für eine Anwendung von Arrays wollen wir ein paar statistische Daten zu den Punkten in einer Klausur ermitteln. Zusätzlich wollen wir die Punkte in einem Array speichern. Da wir noch nicht genau wissen wie viele Studierende an der Klausur teilnehmen, es aber sehr unwahrscheinlich ist, dass es mehr als 100 sind, legen wir eine Array für 100 Noten an:

```
int[] scores = new int[MAX_SIZE];
```

Wir bitten den Nutzer die Noten einzugeben. Dafür können wir wieder den Loop-and-a-Half verwenden. Damit wir wissen wann wir fertig sind, vereinbaren wir, dass die Eingabe der "-1" (dem Sentinel) bedeutet, dass alle Noten eingegeben wurden. Das ist also unser Abbruchkriterium.

Die statistischen Daten die wir ermitteln wollen sind: Anzahl der Klausuren, der Durchschnitt, die niedrigste Punktzahl und die höchste Punktzahl.

FlippedImage

Arrays sind bestens geeignet um mit Bilder zu arbeiten. Wir haben ja oben schon gesehen wie wir auf die Pixel zugreifen können. In diesem Beispiel wollen wir ein gegebenes Bild spiegeln. Das kann horizontal oder vertikal sein. Dazu kreiiren wir ein neues Array

```
int[][] arrayFlipped = new int[height][width];
```

und verwenden zwei verschachtelte Schleifen

```
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        int pixel = array[i][j];
        arrayFlipped[height - i - 1][j] = pixel;
    }
}
```

und die Pixel zu tauschen. Aus dem neuen Array machen wir dann ein neues Bild via

```
GImage flippedImage = new GImage(arrayFlipped);
```

GrayImageXOR

Die Steganographie ist die Kunst der verborgenen Übermittlung von Informationen [3]. Interessant ist, dass man das ganz einfach mit der XOR Funktion, also dem Exklusiven Oder, machen kann. Wie machen wir das? Nehmen wir an wir haben zwei Bilder und deren Pixel Arrays:

```
int pixel1 = array1[i][j];
int pixel2 = array2[i][j];
```

dann holen wir uns jeweils z.B. den Rot-Wert:

```
int r1 = Gimage.getRed(pixel1);
int r2 = GImage.getRed(pixel2);
```

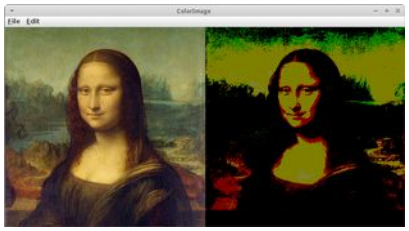
und genauso wie wir diese beide Werte z.B. addieren könnten, können wir auch die XOR Funktion '^' anwenden:

```
int xx = r1 ^ r2;
```

also, wir machen ein bitweises XOR der Bits von r1 mit denen von r2. Wenn wir als Bild-Beispiele das *Taj* und die *Mona Lisa* nehmen, dann kommt da eine lustige Mischung heraus auf der man weder das eine noch das andere erkennen kann. Interessant wird es wenn wir die Pixel dieses Mischlingwerkes nehmen und nochmal die XOR Funktion darüber laufen lassen: dann kommt nämlich wieder das ursprüngliche Bild zum Vorschein. Interessanterweise genau das Gegenstück.

Auf dem gleich Prinzip basiert auch das RAID-5 System das für die Ausfallsicherheit von Festplatten sorgt [4].

ColorImage

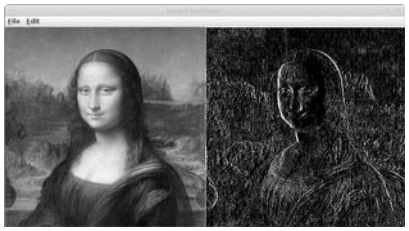


Eine häufige Anwendung der Bildmanipulation ist die Reduzierung der Farben in einem Bild. Das ist eine schöne Anwendung für die Ganzzahl Division (Integer Division).

```
int r = Gimage.getRed(pixel);
r = (r / FACTOR) * FACTOR;
```

Ursprünglich kann *r* ja Werte zwischen 0 und 255 annehmen. Wenn wir diese Zahl durch z.B. 64 teilen, dann haben wir nur noch Werte zwischen 0 und 3. Multiplizieren wir das wieder mit 64, so haben wir nur noch die Werte 0, 64, 128 und 192. Also es gibt nur noch vier Rotwerte.

ImageFilterSimple

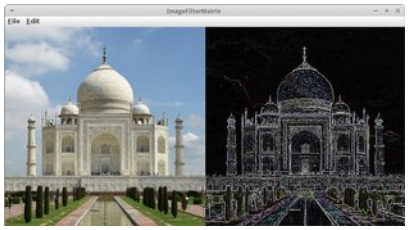


Man kann auch viele andere Bildmanipulationen vornehmen. Z.B. kann man benachbarte Pixel einfach subtrahieren:

```
int r01 = GImage.getRed(array[i][j - 1]);
int r11 = GImage.getRed(array[i][j]);
// difference
int xx = r11 - r01;
xx *= 10;
edge[i][j] = GImage.createRGBPixel(xx, xx, xx);
```

Das Resultat entspricht einer einfachen Kantenerkennung.

ImageFilterMatrix



Viel interessantere Bildmanipulationen werden auf einmal möglich wenn man sich bewusst wird, dass Arrays eigentlich Matrizen sind. Das darf man nicht so laut sagen, sonst wird man verbrannt [6]. Aber wenn man das weiß, dann kann man ganz coole Sachen mit Bildern machen. Bei den Filtern sharpen, blur, edgeEnhance, edgeDetect, oder emboss wie sie aus jedem Bildbearbeitungsprogramm bekannt sind, handelt es sich eigentlich nur um die Anwendung einer Faltungsmatrix [5]. Z.B. sieht die Matrix um ein Bild schärfer zu machen folgendermaßen aus:

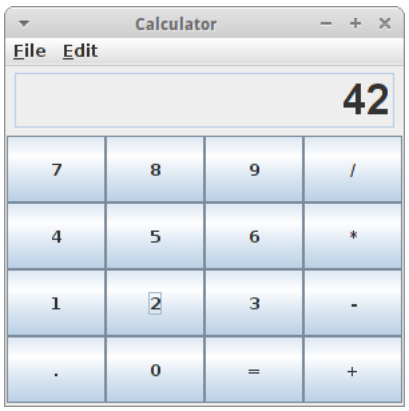
```
private int[][] currentFilter = {
    {  0, -1,  0 },
    { -1,  5, -1 },
    {  0, -1,  0 }
};
```

Das Ausführen der Matrixmultiplikation (also der Anwendung des Filters auf das Bild) erledigt dann folgende Methode:

```
// int alpha = (color >> 32) & 0xFF;
// int red   = (color >> 16) & 0xFF;
// int green = (color >> 8) & 0xFF;
// int blue  = color & 0xFF;
private int applyFilterToPixel(int x, int y) {
    int r = 0;  int g = 0;  int b = 0;
    for (int i = 0; i <= 2; i++) {
        for (int j = 0; j <= 2; j++) {
          r+=((array[x + i][y + j] >> 16) & 0xFF) *currentFilter[j][i];
          g+=((array[x + i][y + j] >> 8) & 0xFF) *currentFilter[j][i];
          b+=((array[x + i][y + j]) & 0xFF) * currentFilter[j][i];
        }
    }
    return Gimage.createRGBPixel(
        checkBounds(r / currentFactor),
        checkBounds(g / currentFactor),
        checkBounds(b / currentFactor));
}
```

die man für jeden Pixel des Ursprungsbildes aufrufen muss. Das Beispiel ist auch deswegen interessant weil man mal eine praktische Anwendung des Rechtsverschiebungs Operators ">>" (right shift) und des bitweisen Und Operators "&" sieht.

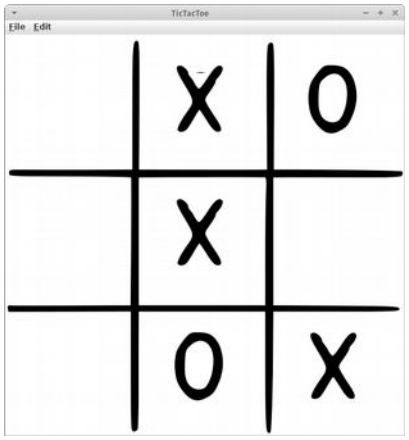
Calculator



Die Anwendungen für Arrays sind wirklich vielfältig. Sehr häufig helfen sie einem ganz viel Code einzusparen. Ein schönes Beispiel ist der Calculator aus dem letzten Kapitel. Man kann natürlich die Knöpfe, also JButtons, alle einzeln erzeugen, man kann das aber auch effektiver machen:

```
private final String[] btnNames =
    { "7", "8", "9", "/", "4", "5", "6", "*",
      "1", "2", "3", "-", ".", "0", "=", "+" };

public void init() {
    setLayout(new GridLayout(4, 4));
    for (int i = 0; i < btnNames.length; i++) {
        JButton btn = new JButton(btnNames[i]);
        add(btn);
    }
}
```

TicTacToeLogic



Arrays können auch für Spiele ganz nützlich sein. Im vierten Kapitel haben wir ja schon die UI für das TicTacToe Spiel geschrieben. Jetzt sind wir soweit auch den Logik Teil zu verstehen. Das Spielfeld kann man nämlich als zwei-dimensionales Array auffassen:

```
private int[][] board = new int[3][3];
```

Ursprünglich sind alle Werte des Spielfelds auf 0 gesetzt. Wenn wir jetzt die Felder die Spieler eins besetzt hat mit einer 1 markieren und die die Spieler zwei besetzt hat mit einer 2 markieren, dann ist das eine perfekte Beschreibung des jeweiligen Spielstandes.

Wenn wir jetzt testen wollen ob ein bestimmter Zug erlaubt ist, dann müssen wir lediglich testen ob der Werte des Spielfelds an der Stelle 0 ist:

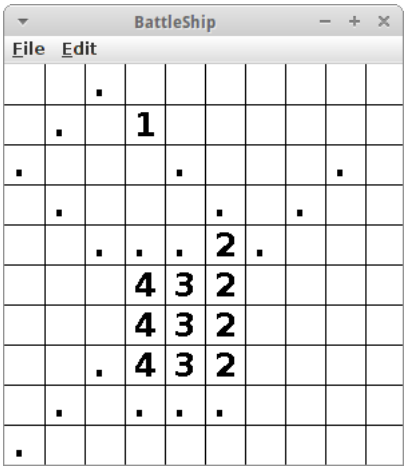
```
public boolean isMoveAllowed(int player, int i, int j) {
    if (board[i][j] == 0) {
        board[i][j] = player;
        return true;
    }
    return false;
}
```

Wenn wir testen wollen ob ein Spieler gewonnen hat, dann müssen wir nachsehen, ob einer der Spieler eine vertikale, horizontale oder diagonale Reihe besetzt hat. Für die vertikale Reihe könnte man das so testen:

```
private boolean checkVerticals() {
    // player 1
    for (int i = 0; i < 3; i++) {
        if ((board[i][0] == 1) &&
            (board[i][1] == 1) &&
            (board[i][2] == 1)) {
            return true;
        }
    }
    // player 2
    for (int i = 0; i < 3; i++) {
        if ((board[i][0] == 2) &&
            (board[i][1] == 2) &&
            (board[i][2] == 2)) {
            return true;
        }
    }
    return false;

}
```

Wir gehen einfach eine Reihe nach der anderen durch (for Schleife) und schauen ob alle drei Werte auf 1 (für Spieler 1) oder 2 (für Spieler 2) gesetzt sind.

BattleShip



Schiffe versenken [7] ist auch ein Spieleklassiker für dessen Umsetzung Arrays sehr nützlich sind. Unser BattleShip Spiel soll ein Spiel Mensch gegen Computer werden, soll heißen der Computer verteilt seine Schiffchen und wir müssen sie finden.

Genauso wie bei TicTacToe verwenden wir für das Spielfeld ein Array von Ganzzahlen:

```
private int[][] board =
    new int[BOARD_SIZE][BOARD_SIZE];
```

Die Schiffe selbst werden durch Zahlen repräsentiert: 5 steht für einen AircraftCarrier, 4 für ein Battleship, 3 für ein Submarine oder einen Destroyer und 2 für ein PatrolBoat. Um festzulegen wie viele es von jeder Schiffsart gibt, können wir auch wieder ein Array verwenden:

```
private final int[] SHIP_SIZES = { 5, 4, 3, 3, 2 };
```

D.h., wenn wir noch ein paar PatrolBoat haben möchten, dann fügen einfach noch ein paar 2er ein.

In der *setup()* Methode

```
private void setup() {
    drawLines();
    initBoard();
    addMouseListeners();

}
```

zeichnen wir das Spielfeld, initialisieren die Boote, und fügen einen MouseListener hinzu. In der *initBoard()* Methode gehen wir einfach durch die Liste von Schiffen (SHIP_SIZES) und fügen eines nach dem anderen mittels der Methode *placeShip(int shipNr, int shipSize)* dem Spielfeld hinzu. Diese Methode kann ganz einfach sein, wenn man die Schiffe einfach nebeneinander platziert, dann wird das Spiel aber ganz einfach, oder sie kann auch sehr kompliziert werden, wenn die Schiffe zufällig verteilt sein sollen. Für uns genügt die einfache Version.

Bleibt nur noch die *mousePressed()* Methode zu implementieren. Wir verwenden wieder unseren Trick mit der Ganzzahl Division:

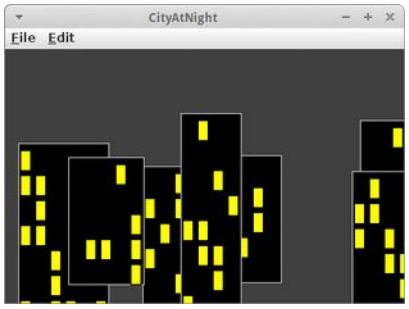
```
public void mousePressed(MouseEvent e) {
    int i = e.getX() / STEP;
    int j = e.getY() / STEP;
    showLabelAt(i, j);
}
```

und es bleibt die *showLabelAt(int i, int j)* Methode zu implementieren. Diese schaut im *board* Array nach ob an der Stelle ein Schiff ist:

```
GLabel lbl = new GLabel("" + board[i][j]);
if (board[i][j] == 0) {
    lbl = new Glabel(".");
}
```

Das ist wieder ein fieser Trick mit dem man sich ein paar unnötige Zeilen Code sparen kann. Den Label zeichnen wir dann einfach an der Position wo die Maus geklickt wurde. Und das war's.

CityAtNight



Wiederverwendung ist ein ganz zentrales Konzept der Objektorientierung. Das kann man sowohl mit Vererbung als auch mit Komposition erreichen. Wir beginnen mit einem Beispiel zur *Komposition*. Erinnern wir uns an Kapitel 2, dort haben wir einen Skyscraper programmiert. Wenn wir jetzt eine ganze Stadt zeichnen möchten, dann wäre es ganz praktisch wenn wir unsere Skyscraper wiederverwenden könnten:

```
public class CityAtNight extends GraphicsProgram {
    private RandomGenerator rgen =
        new RandomGenerator();

    public void run() {
        for (int i = 0; i < 8; i++) {
            int cols = rgen.nextInt(4, 6);
            int rows = rgen.nextInt(4, 8);
            GSkyscraper h = new GSkyscraper(rows, cols);

            int x = rgen.nextInt(0, getWidth() - 40);
            int y = rgen.nextInt(getHeight() / 4, getHeight()/2);
            add(h, x, y);
        }
    }
}
```

Also bräuchten wir eine Klasse GSkyscraper die einen Skyscraper zeichnet. Da ein Skyscraper aus mehreren GRects besteht macht es Sinn, ähnlich wie beim GSmiley, das Ganze als GCompound aufzuziehen:

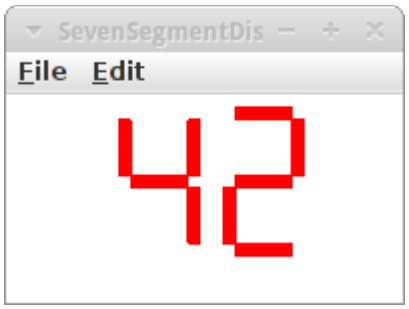
```
public class GSkyscraper extends GCompound {
    ...
}
```

Wie bei jeder Klasse benötigen wir einen Konstruktor

```
    public GSkyscraper(int rows, int cols) {
        ...
    }
```

in dem wir die Anzahl der Fensterreihen und -spalten übergeben. Je nachdem ob alle Skyscrapers gleich aussehen sollen oder unterschiedlich muss man dann noch etwas Zufall in die *addWindow()* Methode einfließen lassen.

SevenSegmentDisplay



Ein weiteres schönes Beispiel für Wiederverwendung mittels *Komposition* ist die Siebensegmentanzeige. Erinnern wir uns an Kapitel 2, dort haben wir eine Siebensegmentanzeige programmiert. Wenn wir jetzt mehrere dieser Siebensegmentanzeigen benötigen, z.B. für einen Zähler, einen Taschenrechner oder ein Uhr, dann wäre es praktisch wenn es eine Klasse *SevenSegmentDisplay* geben würde, die wir einfach mehrmals verwenden könnten, ähnlich einem GRect.

Da eine Siebensegmentanzeige aus mehreren GRects besteht macht es Sinn, ähnlich wie beim GSmiley, das Ganze als GCompound aufzuziehen:

```
public class SevenSegmentDisplay extends GCompound {
    ...
}
```

Wie bei jeder Klasse benötigen wir einen Konstruktor

```
public SevenSegmentDisplay(int width, int height, int ledWidth) {
    ...
}
```

in dem wir idealerweise die Breite und Höhe der Anzeige, sowie die Breite der LEDs vorgeben. Der Konstruktor sollte dann das Display aus GRects konstruieren.

Wirklich praktisch wäre dann noch eine *displayNumber(char c)* Methode,

```
public void displayNumber(char c) {
    turnAllSegmentsOff();
    switch (c) {
    case '0':
        int[] code0 = { 1, 1, 1, 1, 1, 0, 1 };
        turnSegmentsOn(code0);
        break;
    case '1':
    ...
    }
}
```

der man einfach eine Ziffer übergibt, und die diese dann anzeigt. Die *turnSegmentsOn()* Methode könnte wie folgt aussehen:

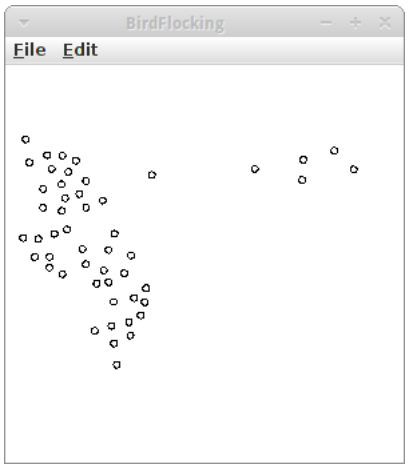
```
private void turnSegmentsOn(int[] code) {
    if (code[0] == 1) {
        upperFrontVertical.setColor(colorOn);
    }
    ...
}
```

Das SevenSegmentDisplay kann man dann ganz einfach in einem GraphicsProgram verwenden:

```
public class SevenSegmentDisplayProgram extends GraphicsProgram {
    public void run() {
        SevenSegmentDisplay ssd1 = new SevenSegmentDisplay(40, 80, 6);
        add(ssd1);
        ssd1.displayNumber('5');
    }
}
```

Erweiterung: Anstelle des JTextField könnte man auch das SevenSegmentDisplay für den Calculator verwenden.

BirdFlocking

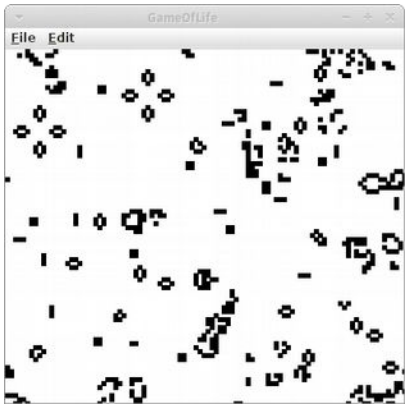


Schwarmverhalten lässt sich bei Fischen, Vögeln und vielen anderen Tieren beobachten. Interessanterweise lässt sich Schwarmverhalten relativ einfach simulieren, die Individuen im Schwarm (auch Boids genannt) müssen lediglich drei einfache Regeln befolgen [11]:

- Separation: halte Abstand von Deinen Nachbarn wenn Du ihnen zu nahe kommst (short range repulsion)
- Alignment: bewege Dich grob in die Richtung Deiner Nachbarn
- Cohesion: bewege Dich grob auf den gemeinsamen Mittelpunkt Deiner Nachbarn zu (long range attraction)

Die Simulation ist ähnlich wie im Planets Projekt, mit dem feinen Unterschied, dass anstelle von Newton's Schwerkraft, die Boid-Regeln gelten.

GameOfLife

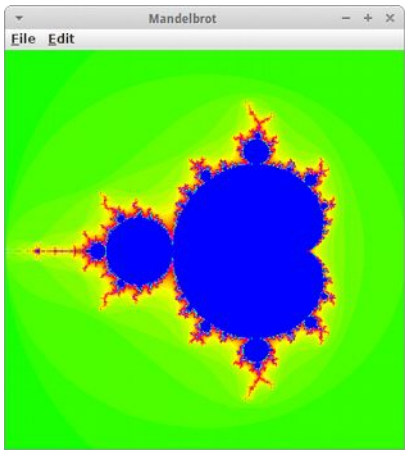


Das größte Genie des letzten Jahrhunderts, John von Neumann, versuchte eine hypothetische Maschine zu konstruieren, die Kopien von sich selbst anfertigen konnte. Dies gelang ihm auch, allerdings hatte das mathematische Modell seiner Maschine sehr komplizierte Regeln. Dem britischen Mathematiker John Horton Conway gelang es Anfang der 70er von Neumanns Ideen drastisch zu vereinfachen, heute bekannt unter dem Namen Conway's *Game of Life* [8].

Das Universum des Spiel des Lebens ist ein zweidimensionales Gitter aus quadratischen Zellen (GRects), von denen jede in einer von zwei möglichen Zuständen sein kann: lebend (schwarz) oder tot (weiß). Jede Zelle hat acht Nachbarn, und abhängig vom Zustand der Nachbarn entscheidet sich der eigene Zustand in der nächsten Runde nach folgenden Regeln:

- jede lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt (Unter-Bevölkerung)
- jede lebende Zelle mit zwei oder drei lebenden Nachbarn lebt
- jede lebende Zelle mit mehr als drei lebenden Nachbarn stirbt (Über-Bevölkerung)
- jede tote Zelle mit genau drei lebenden Nachbarn wird eine lebende Zelle (Fortpflanzung)

Mandelbrot



Die Apfelmännchen sind nach dem französischen Mathematiker Benoît Mandelbrot benannt. Es handelt sich dabei um sogenannte Fraktale, aber die meisten Leute finden sie einfach nur hübsch [9]. Die mathematische Gleichung die hinter der Mandelbrot Menge liegt ist sehr einfach:

```
z_n+1 = z_n * z_n + c
```

dabei sind *z* und *c* komplexe Zahlen. Es handelt sich hier um eine Iteration, d.h. wenn wir *z_n* kennen, dann können wir *z_n+1* ausrechnen. Die Anfangsbedingungen lauten, dass *z_0* gleich null sein soll und *c* ist der Punkt in der komplexen Ebene für den die Farbe ausgerechnet werden soll. Also wenn wir in x- und y-Koordinaten denken, dann ist

```
c = x + i y
```

die Anfangsbedingung. Alles was noch nötig ist, ist das Abbruchkriterium, wann sollen wir mit der Iteration aufhören? Entweder wenn z*z >= 4 ist oder wenn die Anzahl der Iterationen größer als ein maximal Wert ist:

```
while ( (x*x + y*y < 4) && (iteration < max_iteration) ) {
    ...
    iteration++;
}
```

Damit das Ganze dann hübsch aussieht, nehmen wir die Anzahl der Iterationen und kodieren sie in Farbe:

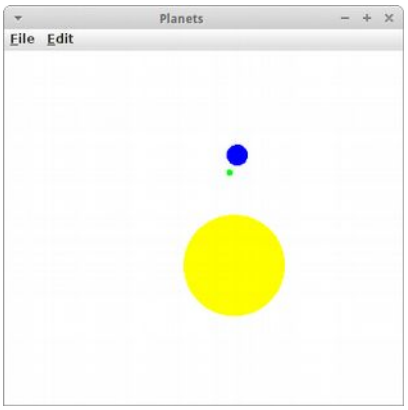
```
int color = RAINBOW_COLORS[iteration % RAINBOW_NR_OF_COLORS];
```

Dabei ist *RAINBOW_COLORS* ein Farbarray, das wir beliebig initialisieren können. Zu guter Letzt brauchen wir noch eine *setPixel()* Methode, die es in der ACM Graphics Bibliothek eigentlich gar nicht gibt. Wir behelfen uns damit, dass wir kleine GRects zeichnen:

```
private void setPixel(double x, double y, Color color) {
    int i = (int) (((x - xMin) * WIDTH) / (xMax - xMin));
    int j = (int) (((y - yMin) * HEIGHT) / (yMax - yMin));
    GRect r = new GRect(1, 1);
    r.setColor(color);
    add(r, i, j);
}
```

Challenges

Planets



Ein schönes Beispiel für Wiederverwendung mittels *Vererbung* ist eine kleine Simulation des Sonne-Erde-Mond Systems. Visuell gesehen, sind Planeten nichts anderes als GOvals. Aber Planeten bewegen sich, d.h. sie haben eine Geschwindigkeit. GOvals haben aber keine Geschwindigkeit. Wir brauchen also ein GOval mit Geschwindigkeit. Genau das ist was Vererbung für uns tun kann:

```
class GPlanet extends GOval {
    public double vx;
    public double vy;

    public GPlanet(int size) {
        super(size, size);
    }

    public void move() {
        move(vx, vy);
    }
}
```

GPlanet ist also ein GOval, hat aber zusätzlich noch eine Geschwindigkeit *vx* und *vy*. Im Konstruktor rufen wir einfach den Konstruktor der Superklasse auf, also den Konstruktor von GOval, und der erzeugt ein GOval mit gegebener Höhe und Breite. Ansonsten benötigen wir lediglich eine *move()* Methode um unseren Planeten zu bewegen.

In unserem Planets GraphicsProgram wollen wir jetzt im setup() drei Planeten erzeugen, also

```
    private void setup() {
        // create sun
        sun = new GPlanet(SUN_MASS);
        sun.setFilled(true);
        sun.setColor(Color.YELLOW);
        sun.vy = SUN_SPEED;
        add(sun, (SIZE - SUN_MASS) / 2, (SIZE - SUN_MASS) / 2);

        // create earth
        ...
        // create earth
    }
```

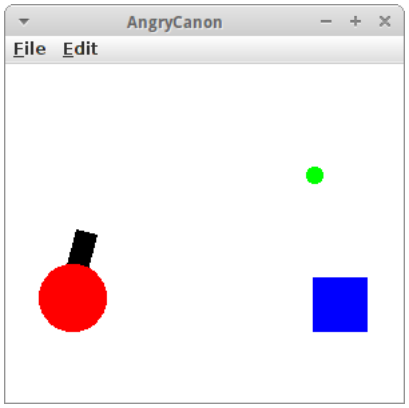
Wir setzen hier den Radius der Sonne gleich der Masse der Sonne. Das ist nicht ganz richtig, für die Simulation aber nicht weiter schlimm. Als nächstes betrachten wir den GameLoop:

```
        while (true) {
            sun.move();
            earth.move();
            moon.move();
            calculateNewVelocities(sun, earth);
            calculateNewVelocities(sun, moon);
            calculateNewVelocities(earth, moon);
            pause(DELAY);
        }
```

Wie üblich im GameLoop, bewegen wir erst die einzelnen Planeten und danach berechnen wir die neuen Geschwindigkeiten. Die Methode *calculateNewVelocities()* sieht etwas kompliziert aus, ist aber nichts anderes als Newton's Gravitationsgesetz.

An diesem Beispiel sieht man sehr schön, dass Simulationen nicht ganz einfach sind: denn nach der zweiten Umkreisung um die Sonne, verlässt uns unser Mond auf Nimmer-Wiedersehen... Schade.

AngryCanon



Unser erstes Projekt das mit Tastatur Events arbeitet wurde von einem populären Spiel mit Vögeln und Schweinen inspiriert. Wie üblich müssen wir die Dinge etwas vereinfachen. Das Ziel ist ein blaues GRect, das wir mit einer Kugel (grünes GOval) treffen sollen. Geschossen wird die Kugel von einer Kanone.

Das schwierigste an diesem Spiel ist die Kanone: denn wir möchten, dass wir ihre Richtung ändern können, dass wir sie drehen können. Es stellt sich heraus, dass nur das GPolygon in der ACM Graphics Library diese Funktionalität bietet, nämlich eine *rotate()* Methode. Wir basteln unsere Kanone, also das Rohr, aus einem GPolygon. Damit das ganze dann hübsch aussieht verstecken wir die "Mechanik" der Kanone hinter einem roten GOval.

In der *setup()* Methode also basteln wir die Kanone, das Ziel ein blaues Rechteck, und wir fügen noch den KeyListener hinzu.

Als nächstes schreiben wir die *keyPressed()* Methode: dort wollen wir abhängig vom KeyCode, die Kanone entweder nach links oder rechts drehen,

```
public void keyPressed(KeyEvent e) {
    int code = e.getKeyCode();
    switch (code) {
    case 37:
        angle += 5;
        canon.rotate(5);
        break;
    case 39:
        angle -= 5;
        canon.rotate(-5);
        break;
    case 32:
        fireBullet();
        break;
    }
}
```

oder wenn der Spieler auf die Leertaste drückt wollen wir die Kugel abfeuern. Wir benötigen den Winkel *angle* als Instanzvariable, damit wir beim Abfeuern die Anfangsgeschwindigkeiten der Kugel setzen können:

```
private void fireBullet() {
    if (bullet == null) {
        vx = -Math.sin(Math.toRadians(angle)) * BULLET_SPEED;
        vy = -Math.cos(Math.toRadians(angle)) * BULLET_SPEED;

        bullet = new GOval(BULLET_SIZE, BULLET_SIZE);
        ...
    }
}
```

Die Bewegung der Kugel selbst wird dann im GameLoop berechnet:

```
while (true) {
    if (bullet != null) {
        moveBullet();
        collisionWithWalls();
        collisionWithTarget();
    }
    pause(DELAY);
}
```

In moveBullet() wollen wir die Kugel bewegen und wir müssen die Schwerkraft wirken lassen:

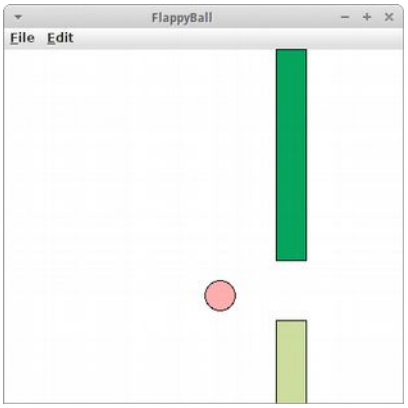
```
private void moveBullet() {
    bullet.move(vx, vy);
    vy += GRAVITY;
}
```

Kommt es zu Kollisionen mit der Wand (also oben, rechts, links oder unten) dann verschwindet die Kugel einfach:

```
remove(bullet);
bullet = null;
```

Kommt es zu Kollisionen mit dem blauen Rechteck, dann verschwindet die Kugel, das Rechteck und das Spiel ist vorbei.

FlappyBall



Auch unser nächstes Projekt ist wieder von einem Spiel mit einem Vogel inspiriert: Allerdings ist unser Vogel ein GOval. Wir können den Vogel mit der Tastatur steuern, genauer der Leertaste. Und der Vogel muss durch ein Hindernis durchfliegen, zwei sich bewegende GRects.

Für das Spiel benötigen wir also einen Vogel (ball) und eine zweigeteilte Wand als Instanzvariablen:

```
private GOval ball;
private GRect upperWall;
private GRect lowerWall;
```

Im *setup()* initialisieren wir diese: Der Ball kommt einfach in die Mitte, und die beiden Rechtecke an den rechten Rand. Die Höhe des Spalts sollte zufällig sein, die Breite des Spalts sollte aber zweimal der Durchmesser des Balls sein. Und natürlich nicht vergessen den KeyListener hinzuzufügen.

Folgt der GameLoop:

```
while (alive) {
    moveBall();
    moveWall();
    checkForCollision();
    pause(DELAY);
}
```

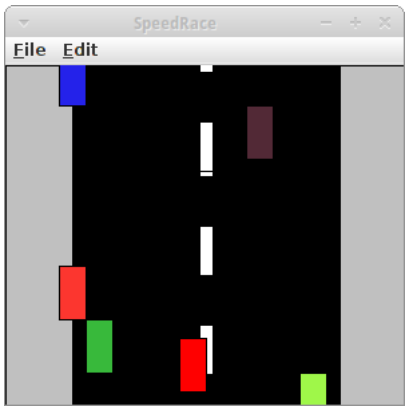
Der Ball bewegt sich nur nach oben oder unten. Normalerweise wirkt die Schwerkraft auf ihn, deswegen fällt er normalerweise nach unten. Die Wand bewegt sich mit konstanter Geschwindigkeit von rechts nach links. Wenn sie den linken Bildschirmrand erreicht verschwindet sie einfach, und eine neue Wand erscheint am rechten Rand.

Was Kollisionen angeht, müssen wir zum einen nach Kollisionen mit der Wand checken: falls es da eine gibt, ist das Spiel vorbei. Bei Kollisionen mit dem Boden, macht es Sinn die Ball-Geschwindigkeit einfach auf 0 zu setzen, und den Ball einfach unten am Bildschirm zu positionieren.

Bleibt noch zu überlegen, was zu tun ist, wenn die Leertaste gedrückt wird? Das ist überraschend einfach:

```
public void keyTyped(KeyEvent e) {
    ballVel = -5.0;
}
```

SpeedRace



Die erste Version von *Speed Race* erschien im Jahr 1974 geschrieben von Tomohiro Nishikado, dem Autor von *Space Invaders* [10]. Es handelt sich salopp gesagt um GTA 0.1, also ein Autorennspiel der ersten Generation. Interessant daran ist wie einfach sich das Gehirn doch überlisten lässt: einfach ein paar weiße Rechtecke die sich von oben nach unten bewegen, schon glaubt man auf einer Straße zu fahren!

Effektiv besteht das Spiel aus einer ganzen Menge Rechtecke. Das erste ist die Straße: die besteht aus zwei Teilen: einem großen schwarzes Rechteck das gar nichts macht und dem Mittelstreifen (middleLane) der aus mehreren (5) weißen Rechtecken besteht, die sich mit konstanter Geschwindigkeit von oben nach unten bewegen.

Es folgt das eigene Auto (car): ein rotes Rechteck, dass sich nur nach links und rechts steuern lässt. Und schließlich die anderen Autos (otherCars), bei denen es sich auch nur um farbige Rechtecke handelt, die sich auch von oben nach unten durch den Bidlschirm bewegen. Im *setup()* werden wie üblich die ganzen Rechtecke angelegt und der KeyListener hinzugefügt.

Der Code für den GameLoop hält sich auch in Grenzen:

```
while (true) {
    moveRoad();
    moveCars();
    checkForCollisionCarsWithWall();
    pause(DELAY);
}
```

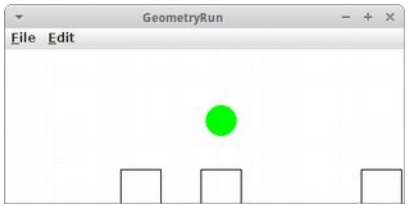
Als erstes bewegen wir die Straße. Das ist eigentlich total trivial, wenn wir uns an unseren Freund den Remainder Operator (%) erinnern:

```
private void moveRoad() {
    for (int i = 0; i < NR_OF_LANES; i++) {
        middleLane[i].move(0, CAR_SPEED);
        double x = middleLane[i].getX();
        double y = middleLane[i].getY() + LANE_LENGTH;
        middleLane[i].setLocation(x, y % SIZE - LANE_LENGTH);
    }
}
```

if-else war gestern. Noch einfacher ist moveCars(), wir bewegen einfach ein Auto nach dem andern. In *checkForCollisionCarsWithWall()* wollen wir eigentlich nur feststellen ob eines der *otherCars* den Bildschirm unten verlassen hat: dann schicken wir es einfach oben wieder auf die Reise, allerdings an einer anderen, zufälligen x-Posiiton.

Bleibt noch die *keyPressed()* Methode: wenn der Spieler auf die linke Pfeiltaste drückt (keyCode = 37), dann bewegen wir das Auto (car) einfach um 5 Pixel nach links, wenn er auf die rechte Pfeiltaste drückt (keyCode = 39), nach rechts um 5 Pixel. Hätte komplizierter sein können.

GeometryRun



Viele von uns haben einen bleibenden Schaden vom Matheunterricht in der Schule davon getragen, deswegen vermeiden wir geometrische Objekte wie die Pest. In diesem Spiel geht es darum, dass wir (ein grünes GOval) unter allen Umständen es vermeiden müssen (mit der Leertaste) mit den ankommenden geometrischen Objekten (GRects) zu kollidieren.

Wie üblich überlegen wir uns zunächst welche Instanzvariablen nötig sind:

```
    private GeometryObstacle[] obstacles;
    private Geometry runner;
```

Bei der *Geometry* Klasse handelt es sich einfach um ein GOval, bei der *GeometryObstacle* um ein GRect. Beide haben aber noch zusätzlich eine Geschwindigkeit:

```
public class Geometry extends GOval implements GeometryConstants {
    public int vx = 0;
    public int vy = 0;

    public Geometry() {
        super(DASH_SIZE, DASH_SIZE);
    }
    public void move() {
        this.move(vx, vy);
    }
}
```

Das mit *extends* haben wir ja schon gesehen, neu ist das *implements*. In diesem Projekt haben wir drei Klassen: *GeometryRun*, *Geometry* und *GeometryObstacle*. Alle drei haben Konstanten, und teilweise die gleichen Konstanten. Damit wir die aber nicht doppelt und dreifach schreiben müssen, fassen wir einfach alle Konstanten in ein *Interface* zusammen:

```
public interface GeometryConstants {
    public final int APP_WIDTH = 400;
    public final int APP_HEIGHT = 200;
    public final int GRAVITY = 2;
    public final int DELAY = 50;

    public final int NR_OF_OBSTACLES = 3;
    public final int OBSTACLES_SIZE = 40;
    public final int OBSTACLES_SPEED = 5;

    public final int DASH_SIZE = 30;
    public final int DASH_JUMP = 20;
    public final int DASH_X_POS = APP_WIDTH / 2;
    public final int DASH_Y_POS = APP_HEIGHT - 2 * OBSTACLES_SIZE;
}
```

Und damit wir die Konstanten benutzen können müssen wir einfach "implements GeometryConstants" an die Klassendeklaration anhängen.

Machen wir weiter mit unserem Spiel, wir beginnen mit dem *setup()*: wir initialisieren den *runner* und die *obstacles*. Den *runner* platzieren wir in die Mitte, die *obstacles* platzieren wir unten am Bildschirm, an zufälligen x-Positionen. Und den KeyListener dürfen wir nicht vergessen.

Der GameLoop ist wieder ganz einfach:

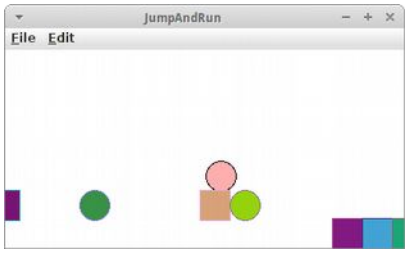
```
    while (true) {
        moveObstacles();
        moveDash();
        checkForCollision();
        pause(DELAY);
    }
```

Die *obstacles* bewegen sich mit konstanter Geschwindigkeit von rechts nach links, und auf den *runner* wirkt nur die Schwerkraft. Die *checkForCollision()* Methode muss einmal dafür sorgen, dass *obstacles* die links verschwinden rechts wieder erscheinen, und sollte Kollision zwischen unserem *runner* und den *obstacles* erkennen.

Bleibt noch die *keyPressed()* Methode: wann immer die Leertaste gedrückt wird, soll die y-Geschwindigkeit des runners einen kleinen Stups erhalten:

```
runner.vy -= DASH_JUMP;
```

JumpAndRun



GeometryRun ist eigentlich ein typisches Jump-and-Run Spiel. Donkey Kong [13] war eines der ersten bekannten Spiele dieses Genres, das im Englischen auch unter dem Begriff "Platform Game" bekannt ist [12]. Was Jump-and-Run Spiele ausmacht ist zum einen, dass es verschiedene Objekte gibt, und zum anderen, dass es verschiedene Levels gibt.

In unserem *JumpAndRun* Projekt haben wir deshalb folgende Instanzvariablen:

```
private GOval ball;
private GObject[] movingObject = new GObject[12];
```

Wir haben also einen *ball* für den Spieler und die *movingObjects* ein Array von GObjects. D.h. das könnten GOval, GRects, oder jedes andere GObject sein. Um welches GObject es sich handeln soll, bestimmen wir mit dem String *world*:

```
private String world = "  RRRR  O   RO  OOO  R";
```

Wenn in dem String ein 'R' steht, soll an der Stelle ein GRect erzeugt werden, für ein 'O' ein GOval, und ansonsten nichts. D.h. mit verschiedenen *world* Strings können wir verschiedene Levels beschreiben. Und ein Level Editor würde nichts anderes machen als diesen String zu editieren.

Was macht die *setup()* Methode? Sie erzeugt den *ball*, fügt den KeyListener hinzu und erzeugt die Welt:

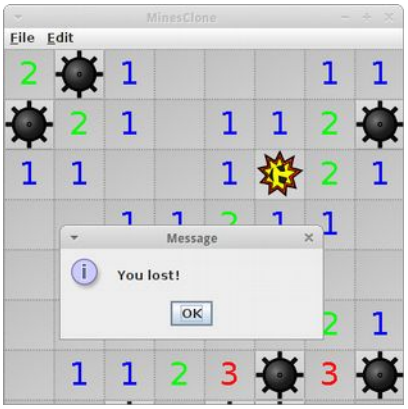
```
private void createNewObjects() {
    for (int i = 0; i < movingObject.length; i++) {
        switch (world.charAt(i)) {
        case 'R':
            GRect rect = new GRect(WIDTH + i * BALL_DIAM,
                                   Y_START, BALL_DIAM, BALL_DIAM);
            rect.setColor(rgen.nextColor());
            rect.setFilled(true);
            rect.setFillColor(rgen.nextColor());
            movingObject[i] = rect;
            add(movingObject[i]);
            break;
        case 'O':
            ...
        default:
            movingObject[i] = null;
            break;
        }
    }
}
```

Der GameLoop ist identisch zum letzten Projekt. Was etwas anders ist, ist die *checkForCollisionWithObjects()* Methode:

```
private void checkForCollisionWithObjects() {
    GObject obj = getElementAt(ball.getX() + BALL_DIAM / 2, ball.getY()
            + BALL_DIAM + 1);
    if ((obj != null)) {
        if (obj instanceof GRect) {
            ballVel = 0.0;
            // ball.setLocation(X_START, HEIGHT - BALL_OFFSET);
        } else {
            alive = false;
        }
    }
}
```

denn abhängig vom Objekt-Typ soll etwas anderes passieren: auf GRects können wir nämlich stehen, aber wenn wir mit GOvals in Berührung kommen, sterben wir.

MinesClone



Wir alle haben schon einmal das Spiel *MineSweeper* (oder *Mines*) gespielt. In dem Spiel geht es darum durch logisches Denken herauszufinden, hinter welchen Feldern Minen versteckt sind [14].

1. Spielfeld

Die erste Frage die wir uns stellen müssen: wie wollen wir das Spielfeld darstellen? Eine Möglichkeit ist ein zweidimensionales Array von chars:

```
private char[][] field =
    new char[FIELD_SIZE][FIELD_SIZE];
```

Warum chars? Weil es dann recht einfach ist darzustellen was sich in der jeweilige Zelle befindet, z.B. könnte ein 'M' eine Mine darstellen und ein Lehrzeichen ' ' bedeutet, dass die Zelle leer ist.

Natürlich müssen wir das Array initialisieren und wir sollten ein paar Minen verteilen. Das tun wir mit der Methode *initialzeField()*:

```
for (int i = 0; i < NUMBER_OF_MINES; i++) {
    int x = rgen.nextInt(0, FIELD_SIZE - 1);
    int y = rgen.nextInt(0, FIELD_SIZE - 1);
    field[x][y] = 'M';
}
```

2. Minen in der Umgebung

Nachdem wir die Minen versteckt haben, müssen wir zählen, wie viele Minen sich in der jeweilige Umgebung einer Zelle befinden. Das können wir natürlich auch selbst, aber ein netter Buchautor hat das schon mal für uns erledigt:

```
MinesHelper.countMines(field);
```

Diese Methode nimmt unser Spielfeld Array als Parameter und verändert es. Das geht deshalb weil Arrays ja als Referenzen übergeben werden (pass-by-reference), also im Original. Jede Zelle (außer Minen) enthält danach die Ziffer die der Anzahl der angrenzenden Minen entspricht:

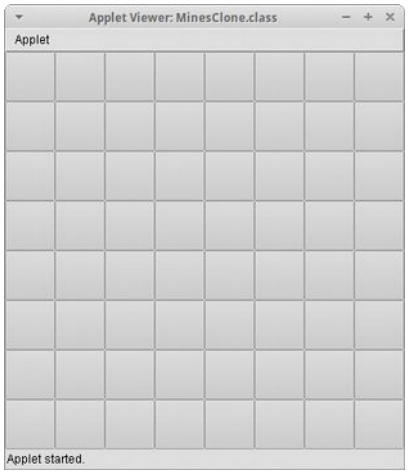
	0	1	2	3	4	5	6	7
0								
1		M			M			M
2								
3					M			
4	M			M	M			
5								
6		M						
7					M			M

vorher

	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	1	M	1	1	M	1	1	M
2	1	1	1	2	2	2	1	1
3	1	1	1	3	M	2	0	0
4	M	1	1	M	M	2	0	0
5	2	2	2	2	2	1	0	0
6	1	M	1	1	1	1	1	1
7	1	1	1	1	M	1	1	M

nachher

3. Spielfeld anzeigen



Nachdem unsere Datenstruktur (das Array) jetzt steht, machen wir mit dem grafischen Teil weiter. Als erstes schreiben wir die *drawInitialField()* Methode. Da am Anfang alle Zellen noch verdeckt sind, ist das ganz einfach, wir zeichnen einfach 8 * 8 "initial.png" Bilder. Das geht ganz einfach mit der *GImage* Klasse:

```
GImage img = new Gimage("initial.png");
add(img, i * PIXEL_PER_TILE, j * PIXEL_PER_TILE);
```

Unser Spiel sieht jetzt dem Original schon sehr ähnlich.

4. MouseEvents

Um auf Klicks der Maustasten reagieren zu können müssen wir natürlich den MouseListener im Setup hinzufügen. Als nächstes implementieren wir die *mouseClicked()* Methode. Wenn die Maus geklickt wurde, müssen wir als erstes herausfinden, auf welche Zelle der Spieler geklickt hat. Hier hilft uns wieder unser alter Freund Ganzzahl-Division:

```
int x = e.getX() / PIXEL_PER_FIELD;
int y = e.getY() / PIXEL_PER_FIELD;
```

Mit diesen Koordination können wir in unserem Array *field[x][y]* nachsehen was sich dort befindet:

```
if (field[x][y] == 'M') {
    ...
} else if (field[x][y] == '0') {
    ...
} else {
    ...
}
```

Wenn der Spieler auf eine Mine geklickt hat, dann ist das Spiel vorbei. Dann könnte man eine Methode *drawWholeField()* schreiben, die das gesamte Spielfeld aufdeckt. Andernfalls, sollten wir die Methode *drawOneTile(x, y)* aufrufen, die an der Stelle x,y das richtige Bild für diese Zelle zeichnet, also das Bild einer Mine ("mine.png") falls es sich um eine Mine handelt, oder das Bild für das leer Feld ("empty.png") überlagert mit einem GLabel, der die Anzahl der angrenzenden Minen anzeigt. Wenn man möchte, könnte man den GLabel auch noch die passende Farbe aus dem LABEL_COLORS[] Array geben.

5. Markieren von Zellen

Ein wichtiger Aspekt fehlt noch in unserem MinesClone: und zwar das Markieren von Zellen als potentielle Minen. Im Original geht das mit der rechten Maustaste. Das ist eigentlich ganz einfach, denn der MouseEvent enthält nämlich die Information welche der Maustasten gedrückt wurde:

```
if (e.getButton() == MouseEvent.BUTTON3) {  ...  }
```

Wenn also der Spieler die dritte Maustaste gedrückt hat, dann soll das Bild "marked.png" an der entsprechenden Stelle gezeichnet werden.

Auch in unserem MinesClone verwenden wir wieder ein Interface (*MinesConstant*) wo wir alle unsere Konstanten speichern.

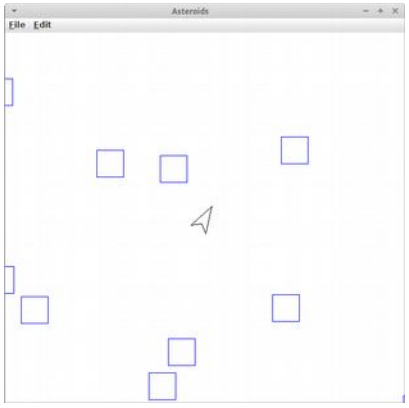
Erweiterungen

Man kann sich noch eine ganze Menge Erweiterungen zu unserem MinesClone denken:

- Wenn der Spieler auf eine Mine klickt, könnte man diese als explodierte Mine zeichnen, um sie von den anderen Minen zu unterscheiden
- Man könnte eine Methode *discoverEmtpyTiles()* schreiben: wenn der Spieler auf ein leere Kachel klickt, dann könnten alle leeren umliegenden Kacheln aufgedeckt werden
- Wenn der Benutzer auf eine Mine klickt, und damit verliert, können Sie die folgenden Zeilen verwenden, um einen Dialog mit dem Benutzer anzuzeigen:

```
IODialog dia = getDialog();
dia.println("You lost!");
```

Asteroids



Laut Wikipedia "ist Asteroids einer der größten Erfolge aller Zeiten in der Geschichte der Computerspiele" [15]. Das soll uns nicht davor abschrecken *Asteroids* selbst zu entwickeln. In dem Spiel geht es darum mit einem Raumschiff durch ein Asteroiden Feld zu fliegen. Und natürlich geht es darum nicht mit den Asteroiden zusammenzustoßen.

1. Vorgefertigte Klassen

Damit wir es schaffen das Spiel in einer vertretbaren Zeit zu entwickeln sind bereits einige Klassen vorgefertigt. Allerdings wenn wir *MarsLander* uns genauer anschauen, dann ist da nichts Neues:

- **GAsteroid:** ist ein GRect mit Geschwindigkeit vx und vy, sowie einer move() Methode.
- **GBullet:** ist ein GOval, ansonsten identisch zu GAsteroid.
- **GSpaceShip:** ist ein GPolygon, genauso wie GAsteroid hat es Geschwindigkeit vx und vy, sowie eine move() Methode. Zusätzlich kann es sich aber drehen, *rotate()*, und beschleunigen via *startEngine()*.

Auch gibt es wieder ein Interface für die Konstanten, *AsteroidConstants*.

In der *setup()* Methode initialisieren wir das Raumschiff, die Asteroiden und fügen den KeyListener hinzu. Das Raumschiff soll in der Mitte des Bildschirms starten. Es soll zehn Asteroiden geben, die blau sein sollen und zufällig verteilt sein sollen. Auch die Geschwindigkeiten sollen zufällig sein:

```
asteroids[i].vx =
    rgen.nextInt(-ASTEROID_MAX_SPEED, ASTEROID_MAX_SPEED);
asteroids[i].vy =
    rgen.nextInt(-ASTEROID_MAX_SPEED, ASTEROID_MAX_SPEED);
```

2. Game Loop

Der GameLoop von Asteroids ist nur geringfügig komplizierter verglichen mit unseren anderen Projekten:

```
public void run() {
    setup();
    waitForClick();
    while (spaceShip != null) {
        moveSpaceShip();
        moveAsteroids();
        moveBullet();
        checkForCollisions();
        pause(DELAY);
    }
    displayGameOver();
}
```

Nach dem *setup()* warten wir bis der Spieler mit der Maus irgendwohin klickt. Dann beginnt das Spiel: in jeder Iteration bewegen wir erst das Raumschiff, dann die Asteroiden, gefolgt von der Kugel, falls eine abgefeuert wurde. Und natürlich müssen wir alle möglichen Kollisionen checken, dazu später mehr.

3. Key Events

Das Raumschiff wird über die Tastatur gesteuert, also müssen wir die *keyPressed()* Methode implementieren:

```
public void keyPressed(KeyEvent e) {
    int code = e.getKeyCode();
    // your code...

}
```

Wenn der Spieler auf die nach oben Pfeiltaste (38) drückt, dann soll das Raumschiff beschleunigen (*startEngine()*), drückt er auf die linke Pfeiltaste (37) soll sich das Raumschiff um 10 Grad nach links drehen, drückt er auf die rechte Pfeiltaste (39), dann soll sich das Raumschiff um 10 Grad nach rechts drehen, also -10 Grad.

Soweit, so gut. Wenn wir unser Spiel jetzt mal kurz antesten, dann sollten die Asteroiden durch die Gegend fliegen, und unser Raumschiff sollte sich drehen können und beschleunigen.

Was noch fehlt ist unsere Selbstverteidigung: wenn wir auf die Leertaste drücken, soll das Raumschiff eine Kugel abfeuern. Also benötigen wir noch einen Eintrag in die *keyPressed()* Methode: wenn der Spieler die Leertaste drückt (' '), dann soll die Methode *fireBullet()* aufgerufen werden. In *fireBullet()* soll also eine neue GBullet erzeugt werden, und zwar an der Position des Raumschiffs, und mit der folgenden Geschwindigkeit:

```
bullet.vx = -Math.sin(Math.toRadians(spaceShip.angle)) * BULLET_SPEED;
bullet.vy = -Math.cos(Math.toRadians(spaceShip.angle)) * BULLET_SPEED;
```

Ein kleiner Test sollte zeigen, dass wir jetzt Kugeln abfeuern können.

4. Kollisionen

Interessant wird das Spiel durch die Kollisionen. Alles in allem gibt es fünf verschiedene:

```
private void checkForCollisions() {
    checkForCollisionAsteroidsWithWall();
    checkForCollisionSpaceShipWithWall();
    checkForCollisionBulletWithWall();
    checkForCollisionBulletWithAsteroid();
    checkForCollisionAsteroidWithSpaceShip();
}
```

Die Kollisionen mit der Wand sind die einfachsten. Sowohl das Raumschiff, als auch die Asteroiden sollen wenn sie den Bildschirm verlassen einfach auf der gegenüberliegenden Seite des Bildschirms wieder erscheinen. Sollte die Kugel den Bildschirm verlassen, dann soll sie einfach verschwinden:

```
remove(bullet);
bullet = null;
```

Um Kollisionen zwischen der Kugel und einem Asteroiden festzustellen, verwenden wir die getElementAt() Methode: falls an der Stelle wo die Kugel ist sich ein GObject befindet, dann muss das ein Asteroid sein. Wir entfernen dann den Asteroiden und die Kugel:

```
remove(obj);
remove(bullet);
bullet = null;
```

Ganz wichtig, wir setzen nicht das *obj* auf *null* (warum?)!

Bleiben noch Kollisionen zwischen Raumschiff und Asteroiden: für das Raumschiff sind die katastrophal, denn die führen zum Ende des Spiels. Wir setzen einfach das Raumschiff auf *null*,

```
remove(spaceShip);
spaceShip = null;
```

und das beendet damit den GameLoop.

Erweiterungen

Man kann sich noch eine ganze Menge Erweiterungen zu unserem Asteroids Spiel denken:

- Game over: wir könnten noch eine Methode *displayGameOver()* schreiben, die einen großen Text (SansSerif-36) in der Mitte des Bildschirms anzeigt.
- Hyperspace: Der Spieler kann auch das Raumschiff in den Hyperraum zu senden, so dass es an einer zufälligen Stelle auf dem Bildschirm wieder erscheint. Natürlich besteht das Risiko, sich dabei selbst zu zerstören wenn man innerhalb eines Asteroiden wieder auftaucht.
- Hübschere Asteroiden: im echten Spiel sind die Asteroiden nicht einfach nur GRects, sondern hübsche GPolygons. Wir müssen eigentlich nur die Klasse *GAsteroid* so modifizieren, dass die Asteroiden wie die im echten Spiel aussehen.
- Asteroiden teilen (schwer): im echten Spiel verschwinden die Asteroiden nicht einfach wenn sie von einer Kugel getroffen werden, sondern sie halbieren sich. Die kleineren Teile bewegen sich dann mit unterschiedlichen Geschwindigkeiten in unterschiedliche Richtungen. Das ist mit einem Array nicht so einfach, wenn man aber eine ArrayList verwendet (nächstes Kapitel), dann ist das gar nicht so schwer.

Fragen

1. Nennen Sie zwei Merkmale einer objektorientierten Sprache.

2. Geben Sie ein Beispiel für Vererbung.

3. Nennen Sie jeweils den Verursacher für die folgenden Events, bzw was passieren muss damit einer dieser Events ausgelöst wird:
   - MouseEvent
   - KeyEvent
   - ActionEvent

4. Beschreiben Sie die Schritte die nötig sind, damit ein GraphicsProgram auf Tastatureingaben (Key Events) reagieren kann.

5. Deklarieren Sie ein Array von Ganzzahlen (int) mit fünf Elementen, das die Zahlen von 1 bis 5 enthält.

6. Vergleichen Sie die beiden Codezeilen:
   ```
   GOval[] circles = new GOval[4];
    GOval o = new GOval(0,0,100,100);
   ```
   Was ist der Unterschied?

Referenzen

Referenzen aus Kapitel 2 bilden auch in diesem Kapitel die Grundlage. Weiter Details zu vielen der Projekte liefert die Wikipedia.

[1] Taj Mahal, Wikipedia, https://en.wikipedia.org/wiki/File:Taj_Mahal_(Edited).jpeg, Author: Yann; edited by Jim Carter, License: Creative Commons Attribution-Share Alike 4.0

[2] Three algorithms for converting color to grayscale, www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

[3] Steganographie, https://de.wikipedia.org/wiki/Steganographie

[4] RAID, https://de.wikipedia.org/wiki/RAID

[5] GNU Image Manipulation Program, Faltungsmatrix, http://docs.gimp.org/de/plug-in-convmatrix.html

[6] Giordano Bruno, https://de.wikipedia.org/wiki/Giordano_Bruno

[7] Schiffe versenken, https://de.wikipedia.org/wiki/Schiffe_versenken

[8] Conways Spiel des Lebens, https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

[9] Mandelbrot-Menge, https://de.wikipedia.org/wiki/Mandelbrot-Menge

[10] Tomohiro Nishikado, Speed Race, https://en.wikipedia.org/wiki/Tomohiro_Nishikado#Speed_Race

[11] Flocking (behavior), https://en.wikipedia.org/wiki/Flocking_(behavior)#Flocking_rules

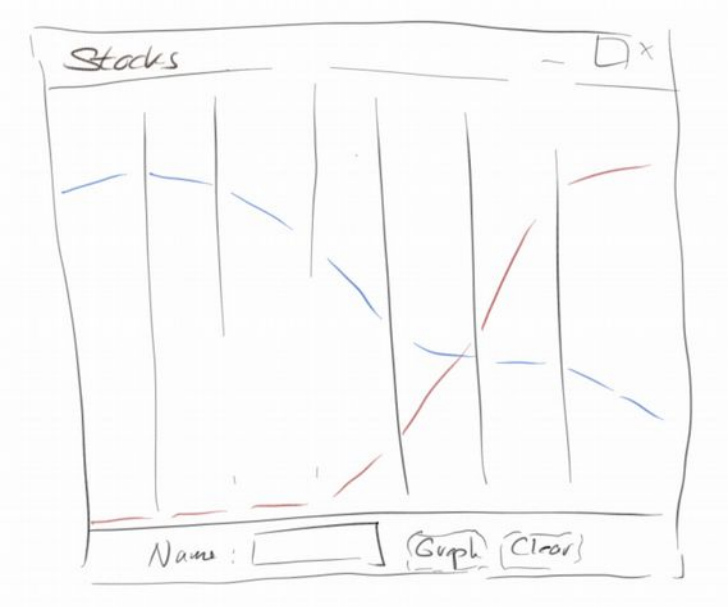
[12] Platform game, https://en.wikipedia.org/wiki/Platform_game

[13] Donkey Kong (Arcade), https://de.wikipedia.org/wiki/Donkey_Kong_(Arcade)

[14] Minesweeper, https://de.wikipedia.org/wiki/Minesweeper

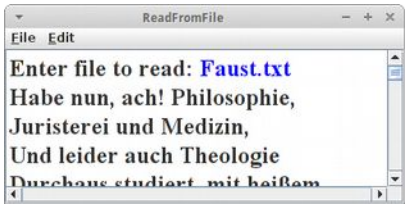
[15] Asteroids, https://de.wikipedia.org/wiki/Asteroids

Stocks



Das Buch neigt sich dem Ende zu, was normalerweise bedeutet, dass es jetzt richtig interessant wird. Wir werden lernen wie wir mit Dateien umgehen, also lesen und schreiben. Dann schauen wir mal was so alles schief gehen kann und wie man das verhindert. Danach erweitern wir unseren Horizont was Datenstrukturen angeht. Die Objektorientierte Analyse kommt als nächstes, und zum Schluss kommen noch Interfaces und Polymorphie.

Files



Sobald wir den Computer ausschalten sind alle Daten futsch. Es sei denn man hat sie gespeichert. Gespeichert heißt man hat sie in eine Datei geschrieben. Das ist eigentlich ganz einfach. Wir fangen mit dem Lesen von Dateien an.

Zum Lesen aus Dateien muss man drei Dinge tun:

1. die Datei öffnen,
2. lesen von der Datei, Zeile für Zeile, und
3. die Datei schließen.

Im Code sieht das dann so aus:

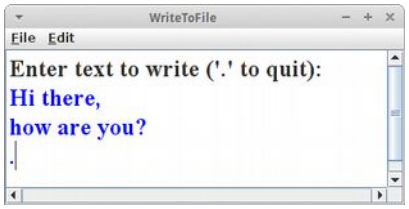
```
// open file
FileReader fr = new FileReader("text.txt");
BufferedReader rd = new BufferedReader(fr);

// read from file, line by line
while (true) {
    String line = rd.readLine();
    if (line == null)
        break;
    println(line);
}

// close file
rd.close();
fr.close();
```

In Java verwenden wir zum Lesen von Textdateien die Klasse *FileReader*. Der übergibt man einfach den Dateinamen der zu öffnenden Datei. Da der FileReader sehr dumm ist, er kann nämlich nur ein Zeichen nach dem anderen lesen, bittet man immer den *BufferedReader* um Hilfe: der kann nämlich ganze Zeilen lesen. Man übergibt den FileReader dem BufferedReader als Argument, damit der BufferedReader weiß welche Datei er denn lesen soll.

Dann benutzen wir einfach einen Loop-And-A-Half um Zeile für Zeile mittels der Methode *readLine()* aus der Datei zu lesen. Wir wissen, dass wir das Ende der Datei erreicht haben, wenn readLine() den Wert *null* zurückliefert. D.h. es gibt nix mehr zu lesen. Danach schließen wir die beiden Readers mit der *close()* Methode.

Leider funktioniert das Programm noch nicht ganz, weil es zu Fehlern (Exceptions) kommen kann. Dazu gleich mehr, aber vorher möchten wir in eine Datei schreiben.

Schreiben



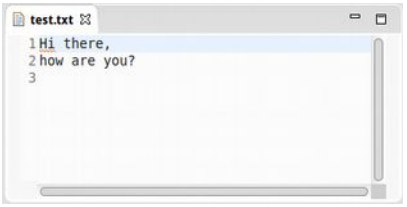
Auch beim Schreiben müssen wir drei Dinge tun:

1. die Datei zum Schreiben öffnen,
2. dann in die Datei schreiben und
3. die Datei schließen.

Schreiben ist einfacher als Lesen, weil wir keinen BufferedDingens brauchen:

```
// open file
FileWriter fw = new FileWriter("test.txt", false);

// write to file, one string at a time
println("Enter text to write ('.' to quit): ");
while (true) {
    String line = readLine("");
    if (line.equals("."))
        break;
```

```
            fw.write(line + "\n");
        }

        // close file
        fw.close();
```



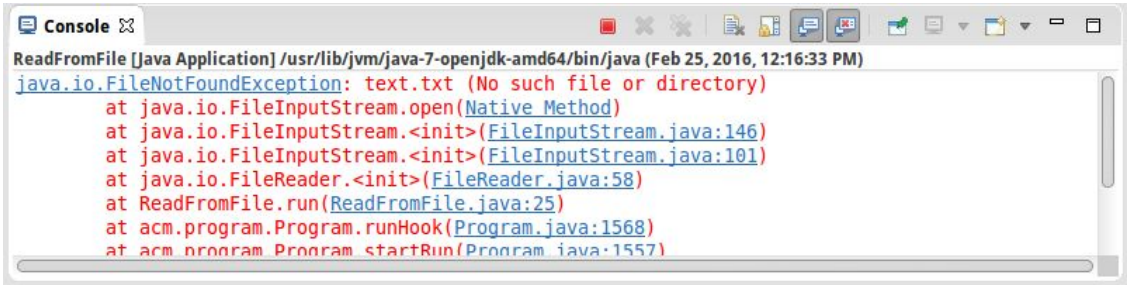
Wir verwenden die Klasse *FileWriter* zum Schreiben von Textdateien. Wir übergeben als erstes den Dateinamen in den wir schreiben wollen, und wir müssen sagen ob wir anhängen wollen oder überschreiben wollen, falls die Datei schon existiert. Wenn wir anhängen wollen, sagen wir "true", falls wir überschreiben wollen sagen wir "false", also nicht anhängen.

Dann benutzen wir wieder den Loop-And-A-Half um Zeile für Zeile mittels der Methode *write()* zu schreiben. Interessant ist vielleicht noch das '\n': es ist das Zeichen für *new line*, und sorgt dafür, dass der nächste String in der "test.txt" Datei in eine neue Zeile geschrieben wird.

Damit wir nicht in einer Endlosschleife hängen bleiben, checken wir ob der Nutzer einen Punkt eingegeben hat, das ist unser Abbruchkriterium. Danach schließen wir unseren Writer mit der *close()* Methode.

Exceptions

Was ist wenn was schief geht? Oder erst mal, was kann denn schief gehen? Es könnte z.B. sein, dass es die Datei die wir zum Lesen öffnen möchten gar nicht gibt. Dann kommt es zu einer *FileNotFoundException*. Eclipse zeigt die sogenannte *StackTrace* im "Console" Fenster:



An der Fehlermeldung erkennt man, dass es die Datei "text.txt" nicht gibt. Die StackTrace sagt einem auch wo denn der Fehler im eigenen Programm ausgelöst wurde. Dazu muss man die Klassen der Reihe nach durchgehen bis man seine eigene findet, in dem Fall oben ist das die *ReadFromFile* Klasse. Und dort steht "ReadFromFile.java:25", und die "25" ist die Zeilennummer in der das Problem auftrat. Wir können aber auch einfach auf den Link klicken, und Eclipse navigiert dann den Cursor an die problematische Stelle.

Kann noch was schief gehen? Ja. Z.B. könnte es sein, dass die Datei zwar existiert, wir sie aber nicht lesen dürfen, da wir keine Berechtigung dazu haben. Dann kommt es auch zu einer FileNotFoundException, aber dieses mal steht in Klammern "Permission denied". Auch beim Schreiben können Sachen schief gehen, z.B. kann es sein, dass wir keine Schreibberechtigung haben, oder es könnte sein, dass wir keinen freien Platz mehr auf der Festplatte haben.

Was passiert denn mit unserem Programm, wenn es zu einer Exception kommt? Es stürzt ab. Das kann sehr unangenehm sein, z.B. wenn das Programm ein Flugzeug steuert in dem wir sitzen, denn dann stürzt auch das Flugzeug ab. Nicht gut.

Try-Catch

Wir müssen also das Flugzeug auffangen, oder besser das Programm. Das machen wir mit *catch*: wir probieren erst mal (*try*) ob alles funktioniert, wenn ja gut, wenn nein, dann sagen wir dem Programm was es tun soll. Im Code sieht das dann so aus:

```
try {
    // code for file access
    ...
} catch ( Exception ex ) {
    // deal with exception
    ...
}
...
```

Wir umgeben also unseren problematischen Code mit einem try-Block. Klappt alles, dann macht das Programm ganz normal weiter (der catch-Block wird nicht ausgeführt). Geht aber was schief, dann bricht das Programm nicht ab, sondern springt in den catch-Block und führt diesen aus. Danach geht das Programm davon aus, dass wir alles unter Kontrolle haben, und führt den Code nach dem catch-Block aus, wie wenn nichts gewesen wäre.

Exceptions treten nicht nur beim Arbeiten mit Dateien auf, sondern auch in vielen anderen Umständen. Betrachten wir kurz die wichtigsten:

NullPointerException: ist wohl die am häufigsten vorkommende Exception. Sie tritt immer dann auf, wenn ein Objekt nicht existiert, also noch keines mit *new* angelegt wurde:

```
GRect fritz = null;
fritz.setColor(Color.RED);
```

ArithmeticException: tritt auf wenn eine mathematische Berechnung nicht möglich ist, der Klassiker, Division durch null:

```
int x = 5 / 0;
```

NumberFormatException: wenn wir versuchen aus einem String eine Zahl zu machen, und der String aber keine Zahl enthält:

```
int x = Integer.parseInt("five");
```

ArrayIndexOutOfBoundsException: wenn wir versuchen auf Elemente eines Arrays zuzugreifen, die es gar nicht gibt:

```
int[] eggs = { 0, 1, 2, 3 };
println( eggs[5] );
```

Data Structures

Bisher kennen wir nur eine Datenstruktur, das Array. Arrays sind zwar o.k., aber so richtig toll sind sie nicht. Viel cooler sind da schon die *ArrayList* und die *HashMap* mit denen wir uns gleich beschäftigen werden.

ArrayList

ArrayListen sind wie Arrays nur schlauer. Fangen wir an eine ArrayList zu deklarieren und zu instantiieren:

```
ArrayList<String> names = new ArrayList<String>();
```

Wir legen also eine ArrayList mit dem Namen "names" an, und in diese ArrayList dürfen nur <Strings> rein, das ist was die spitzen Klammern sagen. Im Prinzip ist es dasselbe wie wenn wir Arrays verwendet hätten:

```
String[] names = new String[10];
```

Allerdings einen Unterschied erkennen wir sofort: bei Arrays mussten wir sagen wie viele Strings wir rein tun wollen, bei der ArrayList ist das nicht notwendig: die ArrayList kann nämlich wachsen und schrumpfen, je nach Bedarf. Das ist super-praktisch!

Machen wir weiter. Wir wollen jetzt ein paar Namen in unsere Namensliste einfügen, allerdings nur, falls der Name noch nicht in der Liste ist. Das geht so:

```
while (true) {
    String name = readLine("Enter new name: ");
    if (!names.contains(name)) {
        names.add(name);
    }
}
```

Wir fragen mit der Methods *contains()* ob ein bestimmter Name schon in der Liste ist. Falls nicht, dann fügen wir ihn mit *add()* hinzu. Wir müssen also auch keinen Index beim Hinzufügen angeben, neue Einträge werden einfach am Ende der Liste eingefügt. Wir können aber auch Einträge irgendwo in der Liste einfügen, dann müssen wir aber den Index angeben.

Wenn wir jetzt unsere Namensliste ausdrucken wollen, dann müssen wir über die Liste iterieren:

```
for (int i = 0; i < names.size(); i++) {
    println(names.get(i));
}
```

Mit der Methode *size()* können wir feststellen wie viele Einträge unsere Liste hat, und mit *get()* lesen wir den Eintrag an einer bestimmten Position. Zusätzlich hat die ArrayList noch folgende praktische Methoden:

- **set():** ersetzt den Eintrag an einer bestimmten Position mit einem neuen
- **indexOf():** sucht nach einem Eintrag und gibt die Position zurück
- **remove():** entfernt einen Eintrag aus der Liste
- **clear():** löscht die komplette Liste.

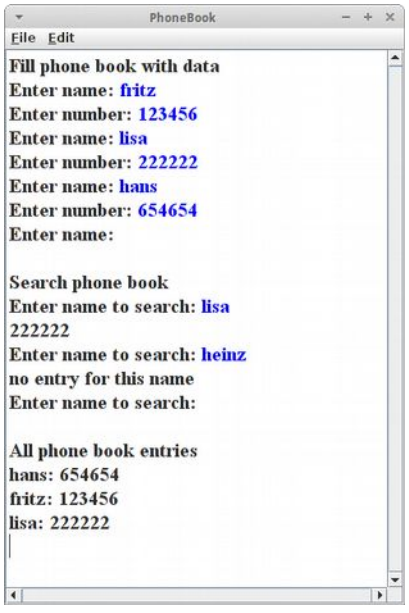
Also, wie wir sehen sind ArrayLists viel besser als Arrays, und deswegen nach langer Zeit wieder mal ein SEP:

SEP: Man sollte immer ArrayList anstelle von Array verwenden (Ausnahme sind Bilder).

HashMap

HashMaps sind wahrscheinlich die praktischste aller Datenstrukturen, weil man in ihnen super suchen kann. Klassische Beispiele sind Telefonbücher und Wörterbücher. Schauen wir uns das Beispiel PhoneBook mal näher an. Wir beginnen mit der Deklaration:

```
HashMap<String, Integer> phoneBook =
    new HashMap<String, Integer>();
```



Das sieht etwas ähnlich wie bei einer ArrayList aus, allerdings sind hier in der spitzen Klammern zwei Datentypen, ein String und ein Integer: <String, Integer>. Das erste ist der Schlüssel (key) und das zweite der Wert (value). Eine HashMap assoziiert also mit einem Schlüssel einen Wert, deswegen nennt man sie manchmal auch assoziative Arrays.

Fügen wir mal ein paar Werte in unser PhoneBook:

```
while (true) {
    String name = readLine("Enter name: ");
    int number = readInt("Enter number: ");
    phoneBook.put(name, number);
}
```

Einfügen geht mit der Methode *put()*. Wir übergeben ihr einfach einen Namen und eine Nummer. Existiert der Name bereits in der Map, dann wird er überschrieben, falls nicht wird er neu eingefügt. In einer HashMap kann es nie zwei Einträge mit gleichem Namen geben, Schlüssel müssen eindeutig sein.

Nachdem wir unser PhoneBook gefüllt haben, wollen wir mal darin suchen, denn das ist ja wofür die HashMap gut sein soll:

```
String name = readLine("Enter name to search: ");
if (phoneBook.containsKey(name)) {
    println(phoneBook.get(name));
} else {
    println("no entry for this name");
}
```

Mit der *containsKey()* Methode können wir testen ob ein Name in der Map existiert, und mit *get()* können wir uns dann die Nummer zu dem Namen geben lassen. Das sieht zwar sehr ähnlich wie bei der ArrayList aus, ist aber viel cooler: bei der ArrayList müssen wir den Index des Eintrags kennen, bei der Map sagen wir einfach: gib mir mal.

Zum Schluß wollen wir mal sehen was in unserer Map drinnen ist:

```
for (String name : phoneBook.keySet()) {
    int number = phoneBook.get(name);
    println(name + ": " + number);
}
```

Das ist jetzt komplett neu: die erste Zeile macht überhaupt keinen Sinn. Das Problem mit den Maps ist, dass es da keine Nummern mehr gibt. Wir können nicht sagen gib mir mal das zweite Element, denn eine Map hat keinen Index. Es gibt auch keine wirklich feste Reihenfolge, d.h. die Reihenfolge in der die Einträge ausgegeben werden, kann komplett anders sein als die Reihenfolge in der sie eingegeben wurden (wie man im Beispiel rechts sieht).

Deswegen der Trick mit der Pseudo-for-Schleife:

```
for (String name : phoneBook.keySet()) {
```

Die heißt soviel wie, gib mir mal einen Namen aus der Liste der Namen in der Map. Und danach, gibst Du mir den nächsten, usw. bis wir alle durch sind. Wenn man das akzeptiert hat, ist der Rest einfach.

Objekt-Orientierte Analyse

Wir befinden uns gerade an einem Übergang: und zwar verlassen wir gerade die prozedurale Ein-Klassen-Welt und betreten die objekt-orientierte Mehr-Klassen-Welt. Der Übergang ist nicht ganz schmerzfrei. Und wo uns bisher der Top-Down Ansatz wirklich gute Dienste erwiesen hat, hilft er uns jetzt nur noch im Kleinen, nicht mehr aber im Großen weiter.

Dafür gibt es jetzt aber die Objekt-Orientierte Analyse, manchmal einfach auch Anforderungsanalyse genannt. Bevor man also anfängt Code zu schreiben, überlegt man sich erst einmal was will man denn überhaupt. Das schreibt man dann am besten in möglichst einfachen Sätzen nieder. Z.B. könnten die Anforderungen für einen Webshop so fomuliert werden:

> "Der Azamon Shop hat Artikel und Warenkörbe. Ein Artikel hat einen Namen, einen Typ und einen Preis. Ein Warenkorb hat einen Benutzernamen und eine Liste von Artikel. Wir können alle Artikel des Shops auflisten. Wir können alle Artikel in einem Warenkorb auflisten. Wir können einen Artikel in den Warenkorb legen. Wir können den Preis aller Artikel in einem Warenkorb berechnen."

Diese Anforderungen sind die Basis für unsere Objekt-Orientierte Analyse.

1.Schritt: In ersten Schritt dieser Analyse nehmen wir einen Farbstift und unterstreichen die Verben grün und die Substantive mit rot:

> "Der Azamon Shop hat Artikel und Warenkörbe. Ein Artikel hat einen Namen, einen Typ und einen Preis. Ein Warenkorb hat einen Benutzernamen und eine Liste von Artikel. Wir können alle Artikel des Shops auflisten. Wir können alle Artikel in einem Warenkorb auflisten. Wir können einen Artikel in den Warenkorb legen. Wir können den Preis aller Artikel in einem Warenkorb berechnen."

2.Schritt: Im zweiten Schritt machen wir eine Liste für die Verben:

- hat
- auflisten
- auflisten
- legen
- berechnen

und die Substantive listen und zählen wir:

- Shop: II
- Artikel: IIIIIII
- Warenkorb: IIIII
- Name: I
- Typ: I
- Preis: II
- Benutzername: I

Das Zählen ist nicht unbedingt notwendig, hilft aber ein bischen zu erkennen was wichtig ist.

3.Schritt: Im dritten Schritt betrachten wir die Liste der Substantive. Welche der Substantive können wir mit einem einfachen Datentyp wie einem *int*, *double*, *String* etc. beschreiben?

- Shop: ???
- Artikel: ???
- Warenkorb: ???
- Name: String
- Typ: String
- Preis: int
- Benutzername: String

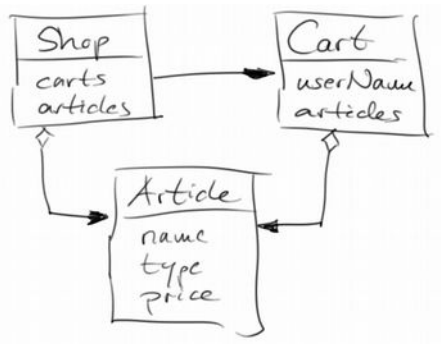
Wenn etwas kompliziert ist, also aus anderen Teilen besteht, wie z.B. Shop, Artikel und Warenkorb, dann handelt es sich um einen komplizierten Datentypen, also eine Klasse. Wir haben also unsere Klassen identifiziert: *Shop*, *Artikel* und *Warenkorb*.

4.Schritt: Im vierten Schritt geht es darum die richtigen Attribute den richtigen Klassen zuzuweisen. Dazu lesen wir einfach noch einmal unsere Anforderungen durch. Dort steht:

- "Der Azamon Shop hat Artikel und Warenkörbe.": also offensichtlich gehören *Artikel* und *Warenkorb* zum *Shop*, sind also Attribute des Shop.
- "Ein Artikel hat einen Namen, einen Typ und einen Preis.": also gehören Namen, Typ und Preis zum *Artikel*.
- "Ein Warenkorb hat einen Benutzernamen und eine Liste von Artikel": also gehören Benutzernamen und Liste von Artikel zum *Warenkorb*.

An dieser Stelle macht es auch Sinn auf Plural zu achten: jedes mal wenn ein Wort im Plural vorkommt, dann bedeutet das, dass wir eine Liste (oder Map) von dem jeweiligen Attribute benötigen. Z.B. aus "Liste von Artikel" oder "Warenkörbe" wird dann eine ArrayList.

Mit der Information können wir schon unsere Klassen mit Attributen hinschreiben:




```
public class Shop {
    private ArrayList<Cart> carts;
    private ArrayList<Article> articles;
}

public class Article {
    private String name;
    private String type;
    private int price;
}

public class Cart {
    private String userName;
    private ArrayList<Article> articles;
}
```

5.Schritt: Beim letzten Schritt geht es darum die Methoden den richtigen Klassen zuzuweisen. Aus den Verben die wir im zweiten Schritt gesammelt haben, werden die Methoden (wir haben ja schon immer gesagt, dass Methoden Tun-Wörter sind). Wir gehen ein Verb nach dem anderen durch.

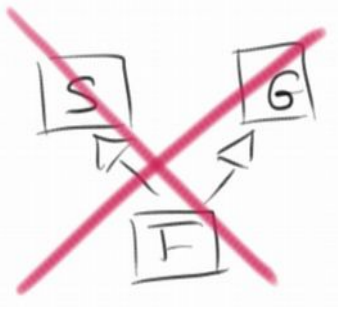
- hat: "hat" zählt nicht als Verb, da es Zugehörigkeit ausdrückt und schon im Schritt 4 verwendet wurde.
- auflisten: bezieht sich auf "Wir können alle Artikel des Shops auflisten", von daher gehört es zum Shop, und sollte *artikelAuflisten()* heißen.
- auflisten: bezieht sich auf "Wir können alle Artikel in einem Warenkorb auflisten", von daher gehört es zum Warenkorb, und sollte *artikelAuflisten()* heißen.
- legen: bezieht sich auf "Wir können einen Artikel in den Warenkorb legen", von daher gehört es zum Warenkorb, und sollte *artikelInWarenkorbLegen()* heißen.
- berechnen bezieht sich auf "Wir können den Preis aller Artikel in einem Warenkorb berechnen", von daher gehört es zum Warenkorb, und sollte *preisAllerArtikelImWarenkorbBerechnen()* heißen.

Und damit stehen unsere Klassen, ihre Attribute und Methoden fest:

```
public class Shop {
    private ArrayList<Cart> carts;
    private ArrayList<Article> articles;

    public void listArticles() {
    }
}

public class Article {
    private String name;
    private String type;
    private int price;
}

public class Cart {
    private String userName;
    private ArrayList<Article> articles;

    public void listArticles() {
    }
    public void addArticleToCart() {
    }
    public void calculatePriceOfArticlesInCart() {
    }
}
```

Was noch bleibt ist sich zu überlegen welche Parameter die Methoden benötigen und was sie als Rückgabewert liefern. Es schadet auch nie, jeder Klasse einen Constructor zu geben. Und natürlich fehlt noch der Code.

Interfaces

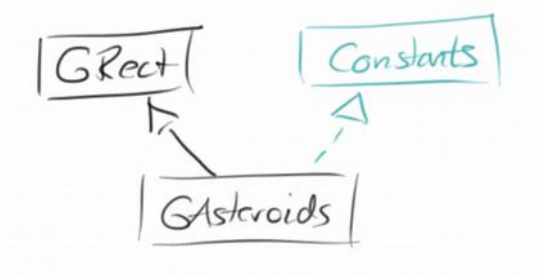


Wir kommen jetzt zum letzten Thema dieses Kapitels: *Interfaces*. Wir haben bereits gehört, dass es in Java keine Mehrfachvererbung gibt, also eine Klasse kann keine zwei Eltern haben. Bisher war das kein Problem, aber jetzt wo wir anfangen größere Programme zu schreiben, stellt sich das schon als ärgerlich heraus. Die Lösung ist, wer hätte es vermutet: *Interfaces*.

Interfaces erlauben uns zwei Dinge zu tun:

- Konstanten zwischen verschiedenen Klassen zu teilen und
- das Definieren einer Schnittstelle zwischen zwei Klassen, daher der Name.

Beginnen wir mit dem ersten. In dem Projekt MinesClone hatten wir ein Interface namens *MinesConstant*:

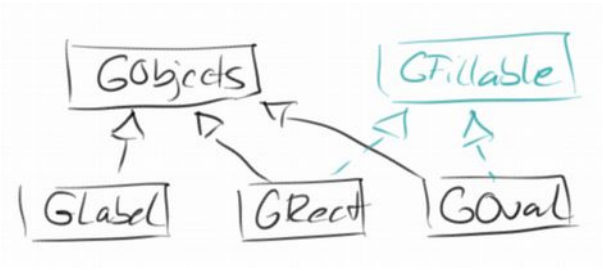



```
public interface MinesConstant {
  /** Playing field should be 8x8 fields
      and 50 pixel per field */
  public static final int FIELD_SIZE = 8;
  public static final int PIXEL_PER_TILE = 50;
     ...
}
```

Wir hatten also alle Konstanten an einer Stelle. Damit diese Konstanten in anderen Klassen verwenden konnten, mussten wir in den Klassen *MinesClone* und *MinesHelper* dieses Interface implementieren:

```
public class MinesClone extends GraphicsProgram implements MinesConstant {
     ...
}

public class MinesHelper implements MinesConstant {
     ...
}
```

Ganz einfach.



Kommen wir zum Definieren einer Schnittstelle zwischen zwei Klassen. Betrachten wir die ACM Grafikklassen: wir haben ja schon gehört, dass GLabel, GRect und GOval Kinderklassen von GObject sind. Zusätzlich gibt es aber noch ein Interface *GFillable*, und die Klassen GRect und GOval implementieren dieses Interface. Für GLabel macht das keinen Sinn, denn was soll denn bitte ein ausgefüllter GLabel sein?

Warum ist jetzt GFillable eine Schnittstelle zwischen zwei Klassen? Betrachten wir unser aller erstes Grafikprogramm BlueRect: Wir wollten ein ausgefülltes Rechteck zeichnen. Da GRect das GFillable Interface implementiert, wissen wir, dass GRect die Methoden *setFilled()* und *setFillColor()* hat. Also die Klasse BlueRect kann sich darauf verlassen, dass es diese beiden Methoden gibt. Umgekehrt verpflichtet sich die Klasse GRect, dass es diese beiden Methoden liefert, da sie ja das Interface GFillable implementiert. Es ist also ein Vertrag zwischen den beiden Klassen BlueRect und GRect. Manchmal wird das auch als "Design by contract (DbC)" [1] bezeichnet.

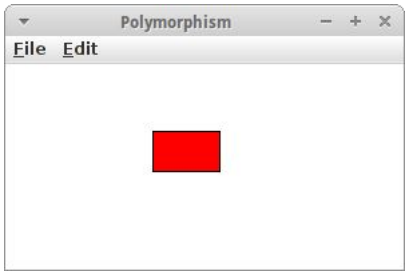
Wie muss jetzt das GFillable Interface aussehen?

```
public interface GFillable {
    public void setFilled(boolean flag);
    public boolean isFilled();
    public void setFillColor(Color c);
    public Color getFillColor();
}
```

Wir sehen also, in der Definition des Interfaces werden Methoden nur angedeutet, aber nicht implementiert. Die Implementierung muss in der jeweiligen Klasse erfolgen, Interfaces dürfen keinen Implementierungscode enthalten.

In der ACM Bibliothek gibt es noch die Interfaces *GResizable*, welches die Klassen GImage, GOval and GRect implementieren, sowie *GScalable*, welches die Klassen GArc, GCompound, GLine, GImage, GOval, GPolygon and GRect implementieren. Wir sehen also, obwohl es in Java nur Einfachvererbung zwischen Klassen gibt, so dürfen Klassen durchaus mehrere Interfaces implementieren.

Polymorphism



Polymorphismus bedeutet Vielgestaltigkeit. Und was hat das bitte mit Java zu tun? Schauen wir uns ein Beispiel an:

```
public class Polymorphism extends GraphicsProgram {

    public void run() {
        GRect frankWalter = new GRect(50, 30);
        add(frankWalter, 50, 50);
        fillWithRedColor(frankWalter);

        for (int i = 0; i < 10; i++) {
            moveByTenPixelToLeft(frankWalter);
            pause(1000);
        }
    }

    public void fillWithRedColor(GFillable rect) {
        rect.setFilled(true);
        rect.setFillColor(Color.RED);
    }

    public void moveByTenPixelToLeft(GObject rect) {
        rect.move(10, 0);
    }
}
```

Das GRect *frankWalter* ist vielgestaltig. Denn gegenüber der Methode *fillWithRedColor()* gibt es sich als GFillable aus, während es sich gegenüber der Methode *moveByTenPixelToLeft()* als GObject ausgibt. Es hat also zwei Gestalten, GFillable und GObject. Mehr ist da nicht.

So und damit sind wir am Ende. Wir kennen jetzt nämlich alle drei Säulen der objektorientierten Programmierung:

- Datenkapselung und Information Hiding
- Vererbung und Komposition
- Polymorphismus

Review

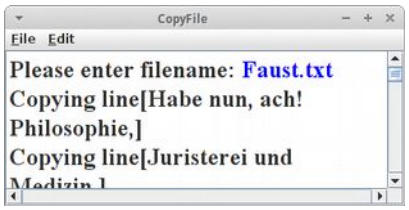
Es wurde wirklich Zeit, dass wir gelernt haben wie man mit Dateien arbeitet. Auch die Fehlerbehandlung hätte eigentlich schon viel früher kommen sollen, aber macht natürlich viel mehr Sinn wenn man mit Dateien arbeitet. Was wirklich neu und sich für die Zukunft als sehr nützlich erweisen wird, sind die Datenstrukturen ArrayList and HashMap. Die werden uns noch so manches Mal das Leben erleichtern. Und wir haben die Themen Interfaces und Polymorphie abgehackt.

Das wichtigste in diesem Kapitel war aber die Objekt-Orientierte Analyse, sozusagen Top-Down 2.0.

Projekte

Hat sich die Arbeit gelohnt? Ich denke die Projekte in diesem Kapitel geben eine klare Antwort: Ja!

CopyFile



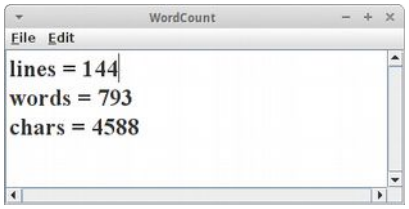
Als kleine Aufwärmübung wollen wir eine Text-Datei kopieren. Der Nutzer soll angeben welche Datei kopiert werden soll. Die Zieldatei soll einfach "copy.txt" heißen.

Im Prinzip gibt es zwei Möglichkeiten dies zu tun: man könnte die gesamten Daten aus der ersten Datei in einen String lesen, und dann in einem Schwung in die Datei"copy.txt" schreiben. Der Nachteil dieses Verfahrens, es ist ungeeignet für sehr große Dateien. Deswegen wollen wir die zweite Möglichkeit umsetzen: Wir öffnen beide Dateien, die erste mit dem FileReader zum Lesen, die zweite mit dem FileWriter zum schreiben. Dann lesen wir Zeile für Zeile, und schreiben jede Zeile sofort in den FileWriter. Wenn wir dann fertig sind müssen wir natürlich beide Dateien wieder schließen.

Minutes

Manchmal sind wir in den Team-Meetings fürs Protokoll zuständig. Dann wäre es ganz praktisch ein Programm fürs Protokoll zu haben. Die Idee ist folgende: man tippt einfach einen Satz, drückt auf *Enter* und die Zeile wird in eine Datei namens "minutes.txt" gespeichert. Man öffnet also eine Datei mittels FileWriter, liest dann kontinuierlich (Loop-and-a-Half) Zeile für Zeile von der Console mittels *readLine()*, und schreibt jede Zeile sofort in die Datei. Natürlich benötigen wir noch ein Abbruchkriterium (Sentinel), das könnte einfach eine leere Zeile sein. Danach dürfen wir natürlich nicht vergessen die Datei zu schließen. Eine nützliche Erweiterung wäre wenn man an jede Zeile die Uhrzeit anfügt, wann der Text eingegeben wurde. Dann würde unser Programm wirklich seinen Namen verdienen.

WordCount



WordCount ist ein ganz einfaches Programm: liest eine bestimmte Datei ein und zählt die Zeilen, die Wörter und die Zeichen die diese Datei enthält. Wir verwenden den FileReader zusammen mit dem BufferedReader um Zeile für Zeile zu lesen. Wir benötigen drei Variablen zum zählen

```
int counterLines = 0;
int counterWords = 0;
int counterChars = 0;
```

(das dürfen lokale Variablen sein).

Die Zeilen zu zählen ist ganz einfach, die Zeichen, auch denn mittels *line.length()* wissen wir wie viele Zeichen in einer Zeile sind. Für das Zählen der Wörter könnte man entweder den StringTokenizer nehmen, oder die Methode *split()* der String Klasse verwenden:

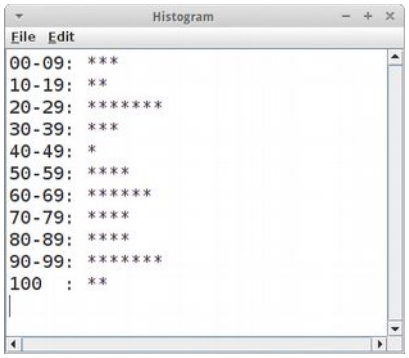
```
private int countWords(String line) {
    String[] words = line.split(" ");
    return words.length;
}
```

Die Methode *split()* ist mit Vorsicht zu genießen. Denn augenscheinlich sieht es so aus, wie wenn sie einfach einen String in seine Einzelteile zerschneiden würde, und diese Einzelteile in einem String-Array speichert. Das tut sie auch, allerdings stellt sich die Frage, nach welchem Kriterium schneidet sie denn den String auseinander? Im Gegensatz zum StringTokenizer, der einfach eine Liste von Trennzeichen nimmt um einen String zu zerteilen, verwendet die *split()* Methode *Reguläre Ausdrücke*. Solange wir also nicht wissen was *Reguläre Ausdrücke* sind sollten wir die *split()* Methode eigentlich nicht verwenden.

CreateFileWithRandomNumbers

Ab und zu (z.B. im nächsten Projekt) benötigen wir eine Datei mit ein paar Zufallszahlen. Das könnten aber auch Zufallsnamen oder Adressen oder was auch immer sein. Der Nutzer soll uns angeben wie die Datei heißen soll in die wir die Zufallszahlen schreiben sollen, dann benötigen wir den Bereich in dem die Zahlen sein sollen, und wir müssten noch wissen wie viele Zahlen wir generieren sollen. Das Programm soll dann mittels FileWriter und RandomGenerator eine solche Datei erzeugen. Für das nächste Projekt benötigen wir Zufallszahlen.

Histogram



Um sich z.B. mal ganz schnell einen Überblick über die Notenverteilung in einer Prüfung zu schaffen, kann man sich ein Histogramm ausgeben lassen. Man kann das natürlich mit einer tollen Grafik machen (kommt später), aber viel einfacher zu programmieren ist das mit einem Konsolenprogramm.

Wir wollen Punkte kumulieren, also zählen wie viele Klausuren Punkte zwischen 0 und 9, zwischen 10 und 19, usw. hatten. Wir könnten z.B. ein Array für elf Zahlen definieren:

```
private int[] histogramData = new int[11];
```

und dann diese mit der Hilfe unseres Freunds Ganzzahldivision befüllen:

```
private void putScoreInHistogram(int score) {
    histogramData[score / 10]++;
}
```

Die Zeile hat es in sich. Nach genügend langer Bewunderung können wir jetzt aber weiter machen.

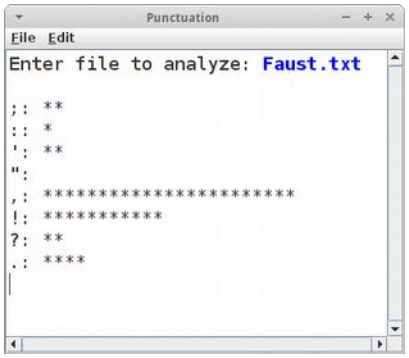
Wir lesen also Zeile für Zeile aus unserer Scores.txt Datei, konvertieren die Strings mittels

```
int score = Integer.parseInt(line);
```

in Ganzzahlen und fügen die dann mittels unserer Wundermethode *putScoreInHistogram()* in unser Array. Wenn wir fertig sind, gehen wir das Array durch und geben es auf der Konsole aus. Wir könnten einfach die Zahlen ausgeben, viel hübscher sind aber kleine Sternchen (Asterisk nicht Asterix!):

```
private String convertToStars(int i) {
    String stars = "";
    for (int j = 0; j < i; j++) {
        stars += "*";
    }
    return stars;
}
```

Punctuation



Wenn wir so im Internet unterwegs sind, dann gibt es Programme (oder besser Algorithmen) die unsere Sprache automatisch erkennen können. Wir tippen ein paar Wörter und Satzzeichen ein, und schon kann uns das Programm sagen welche Sprache das ist. Wir können so etwas ähnliches auch ganz einfach bewerkstelligen.

Die Idee ist es die Satzzeichen, also ";:'",!?.", zu zählen. Das machen wir genauso wie im letzten Projekt (Histogram) und zeigen dann die Häufigkeit der Satzzeichen in einem Histogramm. Es stellt sich heraus, das jede Sprache ihre Lieblingssatzzeichen hat, und man an der Verteilung der Satzzeichen eine Sprache erkennen kann, natürlich muss der zu analysierende Text lang genug sein, und sollte typisch sein (Ulysses wäre also eher ungeeignet [2]).

Der Clou bei diesem Programm ist die Instanzvariable *punctuation*:

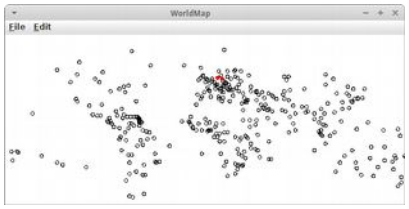
```
private String punctuation = ";:'\",!?.";
```

also ein String der alle möglichen Satzzeichen enthält. Wir haben hier das Zeichen '\"' speziell markiert, denn es ist die einzige Möglichkeit wie man das Anführungszeichen in einen String bekommt. Wir lesen dann wieder Zeile für Zeile aus einer Datei die uns der Nutzer genannt hat, und analysieren jede Zeile

```
private void analyzeForPunctuation(String line) {
    for (int i = 0; i < line.length(); i++) {
        char c = line.charAt(i);
        if (punctuation.contains("" + c)) {
            int index = punctuation.indexOf(c);
            histogramData[index]++;
            totalNrOfPunctuations++;
        }
    }
}
```

Das ist ziemlich schwerer Tabak, aber eigentlich auch wieder nicht. Wir gehen ein Zeichen nach dem anderen durch. Sehen nach ob es ein Satzzeichen ist, und falls ja, erhöhen wir den Zähler für dieses Satzzeichen um eins. Außerdem haben wir noch einen Zähler für die Gesamtzahl der Satzzeichen, das erlaubt uns dann beim Anzeigen der Sternchen die Zahlen zu normieren.

WorldMap



Die Wikipedia hat eine Liste mit den Längen- und Breitengraden vieler Städte weltweit [3]. Wenn wir diese Liste, "Cities.txt", durchgehen und für jede Stadt ein GOval malen, dann können wir eine Weltkarte zeichnen. Wir verwenden also wieder die FileReader/BufferedReader Kombo um Zeile für Zeile aus der Datei zu lesen. Die Daten in einer Zeile

```
Germany, Berlin, 52, 32, N, 13, 25, E
```

müssen wir *parsen*, d.h. die Information so umformen, dass sie für uns nützlich ist. Für unsere Zwecke sind das Land und der Name der Stadt nutzlos. Mit der Genauigkeit nehmen wir es auch nicht so genau, deswegen genügt uns die Grad Angabe, also die *52*, z.B.. Was wir allerdings noch benötigen ist ob sich die Stadt im Norden oder im Süden der Erdkugel befindet:

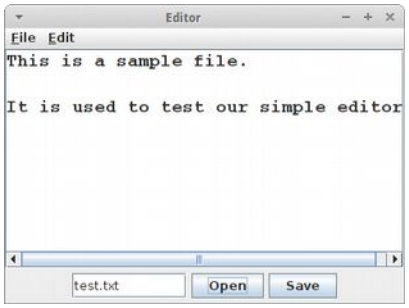
```
String[] data = line.split(",");
String lat1 = data[2].trim();
String lat3 = data[4].trim();

int lat = Integer.parseInt(lat1);
if (lat3.endsWith("S")) {
    lat = -lat;
}
```

und das Gleiche für die Länge (longitude). Das Ganze skalieren wir noch, damit es in unseren Screen passt, also

```
double x = (0.5 - lon / 360.0) * WIDTH;
double y = (0.5 - lat / 180.0) * HEIGHT;
GOval city = new GOval(CITY_SIZE, CITY_SIZE);
add(city, x, y);
```

Sieht doch sehr hübsch aus, oder?

Editor

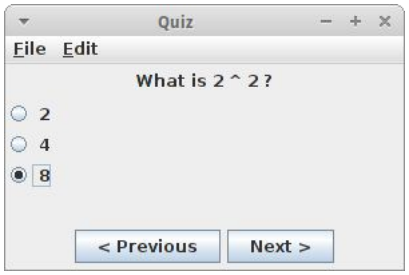


Im Swing Kapitel haben wir ja schon eine UI für einen Editor geschrieben. Endlich können wir auch Dateien lesen und schreiben. In der Methode *saveFile()* holen wir uns einfach den Text aus der JTextArea *display* mittels

```
String text = display.getText()
```

und schreiben den Text in einem Stück mittels eines FileWriters in eine Datei. In der Methode *openFile()* lesen wir Zeile für Zeile aus der Datei und zeigen sie dann in der JTextArea *display* mittels *display.setText(text)* an.

Quiz



Auch für unser Quiz Programm steht die UI bereits. Allerdings ist ein Quiz mit nur einer Frage nicht besonders nützlich. Außerdem sollte man verschiedene Quizze mit unserem Programm halten können.

Wir beginnen mit den Fragen. Die sollten aus einer Text Datei kommen, dann kann man nämliche beliebige Quizze halten. Für unser Beispiel ist das einfach eine Textdatei namens "Quiz.txt":

```
Correct:  1 + 1 = 2 ?; Yes; No; Maybe
What is 2 ^ 2 ?; 2; 4; 8
A zebra has stripes?; Yes; No
```

Jede Zeile entspricht einer Frage mit den möglichen Antworten. Die Frage ist an erster Stelle, gefolgt von den möglichen Antworten. Frage und Antworten durch Strichpunkte getrennt. Diese Datei können wir Zeile für Zeile einlesen, und mittels der *split()* Methode kommen wir an Frage und Antworten:

```
String[] words = line.split(";");
```

Aus jeder Zeile machen wir ein Frage:

```
class Question {
    String question;
    String[] answers;

    public Question(String[] words) {
        question = words[0];
        answers = words;
    }
}
```

indem wir einfach dieses Array an den Konstruktor übergeben. Da die Klasse *Question* nur in unserer Quiz Klasse verwendet wird, kann es eine lokale Klasse sein, muss es aber nicht.

Nun ist die Frage, was machen wir mit den Fragen? Wir stecken sie alle in eine ArrayList:

```
private ArrayList<Question> questions;
```

Man könnte auch ein Array verwenden, aber dann müsste man im Voraus wissen wie viele Fragen ein Quiz hat, das wissen wir aber nicht. Zusammengefasst sieht das dann so aus:

```
while (true) {
    String line = rd.readLine();
    if (line == null)  break;
    String[] words = line.split(";");
    Question q = new Question(words);
    questions.add(q);
}
```

Kommen wir zum Darstellungsteil unseres Programms. Wir benötigen natürlich eine *actionPerformed()* Methode für den *Previous* und *Next* JButton:

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == btnNext) {
        currentQuestion++;
        currentQuestion = currentQuestion % questions.size();
        setQuestion(currentQuestion);
    } else {
        currentQuestion--;
        ...
    }
}
```

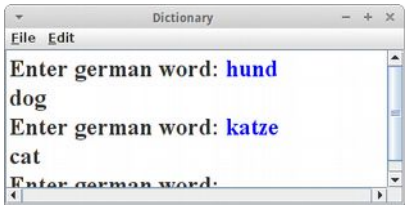
Wir haben also einen Zähler, *currentQuestion*, der auf die momentan aktuelle Frage zeigt, also der Index in der ArrayList *questions*. Was noch bleibt ist die *setQuestion()* Methode.

```
private void setQuestion(int index) {
    Question q = questions.get(index);
    lbl.setText(q.question);
    buildMultipleChoiceAnswers(q.answers);
}
```

Die holt einfach die momentan aktuelle Frage aus der ArrayList, und setzt unser Label mit der Frage und ruft unsere alte *buildMultipleChoiceAnswers()* Methode mit den Antworten auf.

Erweiterungen: Was noch fehlt ist dass die Antworten des Nutzers auch irgendwo gespeichert werden, und am Ende ausgewertet werden. Das ist nicht weiter schwer, braucht aber seine Zeit.

Dictionary



Wörterbücher sind eine typische Anwendung für eine HashMap. Als Beispiel wollen wir ein deutsch-englisches Wörterbuch implementieren. Das Wörterbuch halten wir als Instanzvariable:

```
private HashMap<String, String> dictionary =
    new HashMap<String, String>();
```

und im ersten Schritt müssen wir das Wörterbuch initialisieren:

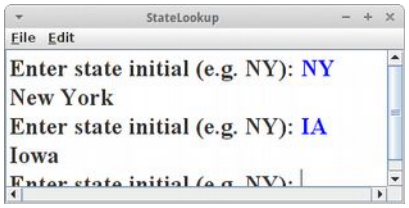
```
private void initialzeDictionary() {
    dictionary.put("hund", "dog");
    dictionary.put("katze", "cat");
    dictionary.put("fisch", "fish");
}
```

Wir fragen dann den Nutzer nach einem deutschen Wort mittels readLine() von der Konsole und der *get()* Methode

```
String english = dictionary.get( german.toLowerCase() );
```

erhalten wir dann das englische Wort. Ganz wichtig, die Übersetzung geht nur in eine Richtung.

StateLookup



Eine andere typische Anwendung für HashMaps ist wenn wir etwas suchen. Z.B. wir haben den Kürzel eines US-Bundesstaates (z.B. "NY") und würden gerne wissen um welchen Bundesstaat es sich denn handelt. Im Internet findet man Tabellen mit der Liste aller Bundesstaaten, z.B. in der Form:

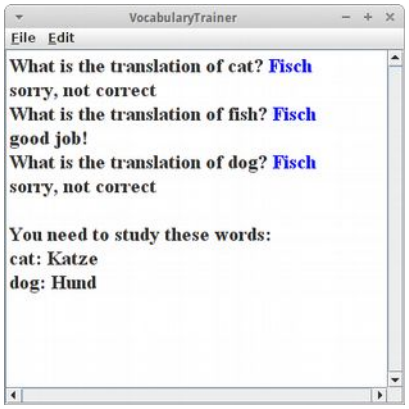
```
AL,Alabama
AK,Alaska
AZ,Arizona
...
```

Wir lesen also Zeile für Zeile, und fügen diese in unsere HashMap ein:

```
private void readStateEntry(String line) {
    int comma = line.indexOf(",");
    String stateInitial = line.substring(0, comma).trim();
    String stateName = line.substring(comma + 1).trim();
    states.put(stateInitial, stateName);
}
```

(das kann man natürlich auch mit der split() Methode oder dem StringTokenizer machen). Der Rest funktioniert dann genauso wie im Projekt Dictionary.

VocabularyTrainer



Das Pendant zu einem Dictionary ist ein Vokabel-Trainer. Allerdings soll unser Vokabel-Trainer für beliebige Sprachen funktionieren, deswegen können wir nicht einfach die Vokabeln hard-coden, sondern sollten diese aus einer Datei lesen. In der Datei "Vocabulary.txt" sind die Vokabeln in folgender Form gespeichert:

```
dog,Hund
cat,Katze
fish,Fisch
...
```

Wir lesen also Zeile für Zeile und speichern die Vokabeln in einer HashMap namens *vocabulary*:

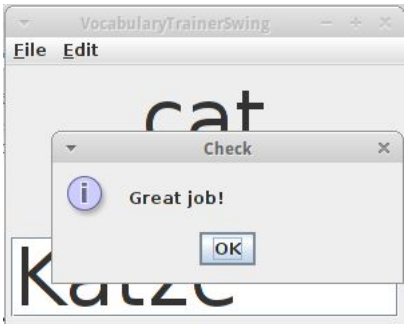
```
private HashMap<String, String> vocabulary =
    new HashMap<String, String>();
```

Was den Vokabel-Trainer jetzt aber vom Dictionary unterscheidet ist, dass er den Nutzer nach der Übersetzung eines englischen Wortes fragt. Ist die Antwort des Nutzers richtig, dann bekommt ein verbales Klopfen auf die Schulter. Wusste der Nutzer allerdings das Wort nicht, so müssen wir es in einer Liste von *unknownWords* speichern:

```
private ArrayList<String> unknownWords = new ArrayList<String>();
```

Am Ende sollten wir dann die Liste der Wörter die der Nutzer nicht wusste auflisten, damit er sie noch mal in Ruhe lernen kann.

VocabularyTrainerSwing



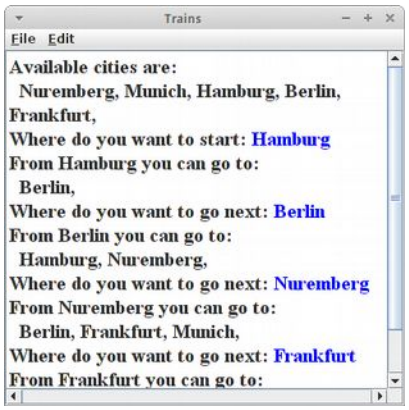
Konsolenanwendungen sind zwar einfach, aber hässlich. Da wir aber die schwere Arbeit schon im letzten Projekt geschafft haben, machen wir daraus einfach eine Swing Applikation. Die besteht aus einem JLabel im Norden und einem JTextField im Süden. Wir zeigen dem Nutzer das zu übersetzende Wort im JLabel, und der Nutzer soll dann die Übersetzung im JTextField eingeben.

Für das Feedback an den Nutzer würden wir gerne eine Dialogfenster verwenden. In Swing geht das mit der *JOptionPane*:

```
public void actionPerformed(ActionEvent e) {
    String english = englishLbl.getText();
    String guess = germanTf.getText();
    if (guess.toLowerCase().equals(dictionary.get(english))) {
        JOptionPane.showMessageDialog(this, "Great job!",
                "Check", JOptionPane.INFORMATION_MESSAGE);
    } else {
        JOptionPane.showMessageDialog(this, "Try again!",
                "Check", JOptionPane.INFORMATION_MESSAGE);
    }

    setRandomWord();
}
```

Ansonsten, können wir den Code vom Projekt VocabularyTrainer wiederverwenden.

Trains



Eine sehr schöne Anwendung für HashMaps sind Fahrpläne. Nehmen wir an wir wollen von München nach Berlin und es gibt aber keine direkte Verbindung. Dann sehen wir im Fahrplan nach und sehen, dass man von München nach Nürnberg fahren kann, und von Nürnberg gibt es einen Zug nach Berlin. So eine einfacher Fahrplan könnte einfach eine Textdatei sein, die alle Verbindung enthält:

```
Nuremberg > Berlin
Nuremberg > Frankfurt
Nuremberg > Munich
Munich > Nuremberg
Hamburg > Berlin
```

Wichtig ist hier, dass die Verbindungen zwischen Städten eine Richtung haben, also der Zug geht von Nürnberg nach Berlin, muss aber nicht zurück gehen.

Der nächste Schritt ist sich zu überlegen wie man so einen Fahrplan in einer HashMap unterbringt. Es ist klar, dass der Key der Ausgangsbahnhof sein muss. Da es aber mehrere Zielbahnhöfe geben kann, müssen wir hier eine Liste verwenden:

```
private HashMap<String, ArrayList<String>> connections;
```

Außerdem macht es auch noch Sinn eine Liste von allen Bahnhöfen irgendwo zu haben:

```
private ArrayList<String> cities;
```

Im *setup()* lesen wir also den Fahrplan und befüllen unsere beiden Datenstrukturen. Mit dem StringTokenizer trennen wir source von destination:

```
StringTokenizer st = new StringTokenizer(line, ">");
String source = st.nextToken().trim();
String destination = st.nextToken().trim();
```

Dann sollten wir checken, ob es den Ausgangsbahnhof schon gibt

```
if (!cities.contains(source)) {
    cities.add(source);
    connections.put(source, new ArrayList<String>());
}
```

und schließlich müssen wir die neue Verbindung hinzufügen:

```
ArrayList<String> cits = connections.get(source);
cits.add(destination);
```

Nachdem die Daten jetzt geladen sind, können wir mit dem eigentlichen Programm fortfahren. Als erstes sollten wir dem Nutzer eine Liste aller Ausgangsbahnhöfe auflisten. Daraus sollte er seinen Ausgangsbahnhof wählen. Im nächsten Schritt listen wir die möglichen Zielbahnhöfe auf, und lassen den Nutzer wieder wählen. Das machen wir so lange, bis der Nutzer seinen Zielbahnhof erreicht hat, also den leeren String eingibt.

Erweiterungen: Was natürlich cool wäre, wenn der Nutzer einfach nur seinen Ausgangsbahnhof und Zielbahnhof eingeben könnte, und das Programm dann automatisch eine Route vorschlägt. Gegen Ende des nächsten Semesters können wir das auch.

Adventure

Auch eine schöne Anwendung für HashMaps sind Abenteuer Spiele. In vielen von diesen textbasierten Spielen geht es darum eine Welt zu erkunden, und Gegenstände einzusammeln. Wir konzentrieren uns hier auf den Erkunden Teil, aber der Einsammel-Teil ist auch nicht so schwer.

Ähnlich wie beim *Trains* Projekt benötigen wir eine Beschreibung der Umgebung. Am einfachsten ist da eine Beschreibung unserer Wohnung. Also bei uns zu hause sieht das so aus:

```
hallway > kitchen
hallway > living room
hallway > bath room
kitchen > hallway
living room > hallway
bath room > hallway
bath room > kitchen
```

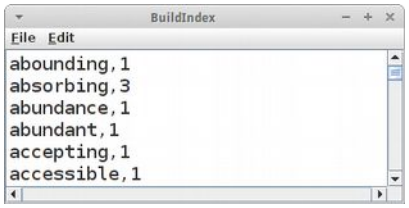
Auch hier verwenden wir wieder eine HashMap die den Plan unserer Wohnung wiedergibt:

```
private HashMap<String, ArrayList<String>> roomMap;
```

Da es sich um ein Erkundungsspiel handelt, benötigen wir keine Liste aller Räume, anstelle lassen wir den Spieler einfach in der Küche mit seiner Erkundung beginnen. Wir listen dann die Räume auf die von der Küche aus zu erreichen sind, und bitten den Spieler eine Wahl zu treffen. Auf diese Art und Weise kann der Spieler nach und nach unsere ganz Wohnung erkunden. Mit der Eingabe des leeren Strings endet das Spiel.

Erweiterungen: Für diesen Spieltyp gibt es zahllose Erweiterungen. Man könnte zum Beispiel die Welt aus StarWars oder Herr der Ringe auf diese Art abbilden. In den verschiedenen Räumen könnte man magische Gegenstände verstecken. Und manche Räume kann man nur betreten wenn man einen bestimmten Gegenstand hat, usw...

BuildIndex



Bücher aus Papier kann man nicht so leicht durchsuchen wie elektronische Bücher. Deswegen haben die meisten Bücher hinten einen Index, auch Stichwortverzeichnis genannt. Als Beispiel wollen wir eine Liste von Stichwörtern für das Buch "TomSawyer.txt" erstellen.

Wie üblich gehen wir Zeile für Zeile durch das Buch und benutzen den StringTokenizer

```
StringTokenizer st =
    new StringTokenizer(line, "[]\"',;:.!?()-/ \t\n\r\f");
```

um die Wörter aus eine Zeile zu extrahieren. Das ist eines der wenigen Male wo die *split()* Methode der String Klasse nicht funktionieren würde (es sei denn man beherrscht Reguläre Ausdrücke).

Wir gehen also alle Zeilen und alle Wörter (Tokens) durch und speichern diese in einer HashMap

```
private Map<String, Integer> words = new HashMap<String, Integer>();
```

Diese HashMap befüllen wir dann mit der folgenden Methode:

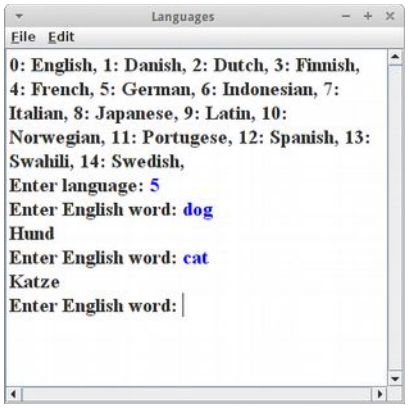
```
private void addWordToHashMap(String word) {
    if (word != null) {
        if (words.containsKey(word)) {
            int count = words.get(word);
            words.put(word, ++count);
        } else {
            words.put(word, 1);
        }
    }
}
```

denn wir wollen zählen, wie häufig ein bestimmtes Wort vorkommt. Jetzt müssen wir nur noch die Map auf der Konsole ausgeben.

Erweiterungen:

- Wenn wir die Liste betrachten, stellen wir fest, dass die meisten Wörter mit weniger als acht Buchstaben eigentlich nichts in einem Index verloren haben. Also sollten wir sie gar nicht in die Liste mit aufnehmen.
- Man könnte noch Wörter die im Plural enden herausfiltern (das ist im Englischen relativ einfach)
- Man könnte Wörter mit nutzlosen Endungen (im Englischen "ly", "ial", "ive", "ous", "ed") herausfiltern.
- Man kann auch eine Liste von Stoppwörtern haben und diese dann herausfiltern [6].
- Sortieren: wenn wir anstelle von HashMap eine TreeMap verwenden, dann ist der Index auf einmal sortiert. Warum das so ist, lernen wir nächstes Semester.

Languages



Was ist schon ein Wörterbuch, das von einer Sprache in eine andere übersetzt? Wir wollen ein Wörterbuch, das von einer Sprache in zehn übersetzt! Dazu muss man aber erst mal irgendwo die nötigen Daten finden. Glücklicherweise gibt es auf dem Website zu dem Buch "Introduction to Programming in Java" von Robert Sedgewick und Kevin Wayne [4] (übrigens ein Super-Buch) eine Datei die die Übersetzungen von über 800 englischen Wörter in zehn andere Sprachen enthält [5].

Die Datei "Languages.csv" enthält diese Daten:

```
"cat","kat","kattekop","kissa","chat, matou, rosse","Katze",...
```

Das erste Wort in jeder Zeile ist das englische Wort, gefolgt von der dänischen, der holländischen, usw., Übersetzung. Welche Sprache an welcher Stelle kommt steht in der ersten Zeile der Datei. Die Daten zu parsen wird nicht ganz einfach, wenn wir uns die französische Übersetzung für Katze ansehen, denn es gibt anscheinend mindestens drei Worte für Katze. Aber wenn wir nach Anführungsstrichen mit der *indexOf()* Methode suchen,

```
private ArrayList<String> parseLine(String line) {
    ArrayList<String> translations = new ArrayList<String>();
    while (true) {
        int begin = line.indexOf("\"");
        if (begin < 0)
            break;
        int end = line.indexOf("\"", begin + 1);
        String s = line.substring(begin + 1, end);
        line = line.substring(end + 1);
        translations.add(s);
    }
    return translations;
}
```

dann ist das durchaus machbar. Die Methode *parseLine()* zerlegt also eine Zeile aus unserer Datei, und wandelt sie in eine ArrayList von Strings um. Diese ArrayList enthält also das englische Wort mit all seinen Übersetzungen. Deutsch ist an sechster Stelle, d.h. mit

```
ArrayList<String> translations = parseLine(line);
String german = translations.get(5);
```

erhalten wir die deutsche Übersetzung des Wortes. So parsen ist erledigt.

Ähnlich wie bei unserem einfach Dictionary Projekt, wollen wir ja nach Wörtern suchen, und dafür verwenden wir die HashMap:

```
private Map<String, List<String>> dictionary;
```

Allerdings speichern wir jetzt eben nicht nur ein Wort pro englischem, sondern zehn, und deswegen steht da auch *List<String>*. Die Liste zu befüllen ist ganz einfach:

```
ArrayList<String> translations = parseLine(line);
dictionary.put(translations.get(0), translations);
```

Das Übersetzen ist jetzt ganz einfach. Wir müssen nur wissen welche Sprache gewünscht ist (also z.B. 5 für Deutsch) und welches Wort übersetzt werden soll:

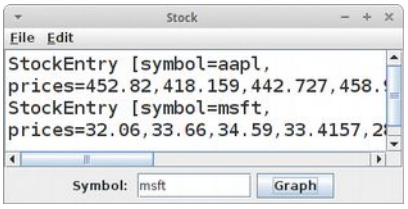
```
private String translate(String english, int lang) {
    List<String> words = dictionary.get(english);
    if (words != null) {
        return words.get(lang);
    }
    return null;
}
```

Das *if* ist nötig um zu verhindern, dass unser Programm abstürzt falls wir nach einem Wort suchen, das nicht in unserer Datenbank ist.

Erweiterungen:

- wie könnte man von jeder der zehn Sprachen in jede andere der zehn Sprachen übersetzen?
- könnte man nicht nur Wörter, sondern ganze Sätze übersetzen?
- man könnte natürlich auch eine hübsche UI Anwendung dafür schreiben.

StockCharter



Kommen wir zu unserem letzten und größten Projekt: *StockCharter*. Es geht darum Aktienkurse grafisch darzustellen. Wie üblich fangen wir aber einfach an.

In einem ersten Schritt schreiben wir ein ConsoleProgram, das im Süden einen JLabel, ein JTextField und einen JButton hat. Wenn der Nutzer dann einen Aktienkürzel (z.B. "msft") im Textfeld eingeben und auf den Knopf klickt, sollen im Konsolenteil die Daten zu der jeweiligen Aktie angezeigt werden. Der UI Teil sollte uns inzwischen keine große Mühe bereiten.

1. Aktienkurse

Kommen wir zu den Daten, den Aktienkursen. Es ist überraschend schwer an freie Daten für Aktienkurse zu kommen, aber glücklicherweise hat die Firma QuantQuote historische Kurse für den Standard & Poor's 500 (S&P 500) Aktienindex zur freien Verfügung gestellt [7]. In diesem Index sind unter anderem Firmen wie Microsoft (msft), IBM (ibm), Ebay (ebay) und Netflix (nflx) gelistet. Die Daten sind in der Datei "SP500_HistoricalStockDataMonthly.csv" zu finden und sehen wie folgt aus:

```
,20130801,20130703,20130605,20130507,20130409,20130311,20130208,...
...
msft,32.06,33.66,34.59,33.4157,28.5002,27.7355,26.9466,26.2568,...
msi,55.07,57.11,56.9164,56.6377,63.0081,62.0605,59.6717,55.9545,...
```

D.h., es beginnt mit dem Kürzel der Firma, gefolgt von den Kursen. Den Datum des jeweiligen Kurses kann man der ersten Zeile entnehmen.

2. Datenbank

Der nächste Schritt ist es die Daten einzulesen. Wir beginnen mit der Klasse *StockDataBase*, die alle Daten enthalten soll, also unsere *Datenbank*. Wir öffnen die Datei "SP500_HistoricalStockDataMonthly.csv":

```
BufferedReader br = new BufferedReader(new FileReader(fileName));
// first line contains dates:
String line = br.readLine();
readDates(line);
// other lines contain data:
readStockPrices(br);
br.close();
```

Die erste Zeile enthält den jeweiligen Datum, d.h. wir verarbeiten diese in der Methode *readDates()* und speichern sie in der Instanzvariable *dates*:

```
private List<String> dates = new ArrayList<String>();
```

Danach lesen wir die ganzen Aktienkurse mit der Methode *readStockPrices()* Zeile für Zeile ein.

```
private void readStockPrices(BufferedReader br) throws IOException {
    String line;
    while (true) {
        line = br.readLine();
        if (line == null)
            break;
        StockEntry entry = new StockEntry(line);
        stockDB.put(entry.getSymbol(), entry);
    }
}
```

Dabei machen wir aus jeder Zeile einen *StockEntry*, und speichern diesen in unserer HashMap die alle Aktienkurse enthält ab:

```
private Map<String, StockEntry> stockDB =
    new HashMap<String, StockEntry>();
```

Ein *StockEntry* enthält alle Daten die zu einer Aktie gehören, und das sind Name und Kurs:

```
public class StockEntry {
    private String symbol;
    private List<Double> prices;
    ...
}
```

Die Klasse *StockEntry* benötigt also einen Konstruktor der eine Zeile parst und die eigenen Instanzvariablen *symbol* und *prices* initialisiert:

```
public StockEntry(String line) {
    String[] sVals = line.split(",");
    symbol = sVals[0];
    prices = new ArrayList<Double>();
    for (int i = 1; i < sVals.length; i++) {
        if (sVals[i].equals("null")) {
            prices.add(-1.0);
        } else {
            prices.add(Double.parseDouble(sVals[i]));
        }
    }
}
```

Außerdem benötigt sie noch eine *getSymbol()* und eine *getPrices()* Methode, und natürlich eine *toString()* Methode wäre auch noch schön.

Kommen kurz zurück zur *StockDataBase* Klasse: der fehlen noch zwei Methoden: *findEntry()* und *getDates()*. Die erste der beiden sucht nach einem gegebenen Symbol (z.B. "msft") und soll den dazugehörigen *StockEntry* aus der HashMap *stockDB* zurückliefern, die zweite soll einfach die Liste mit den Dati zurückgeben.

3. Console

Um zu testen, dass unsere Datenbank richtig funktioniert, fügen wir einen ActionListener zu unserem ConsoleProgram hinzu und implementieren die *actionPerformed()* Methode:

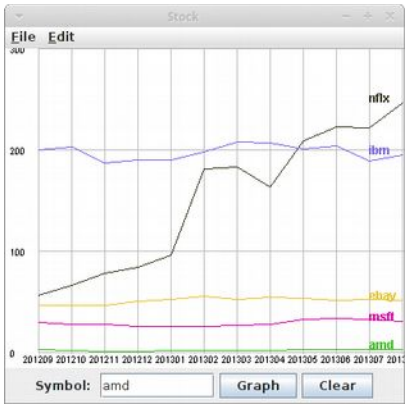
```
public void actionPerformed(ActionEvent e) {
    StockEntry entry = db.findEntry(tfSymbol.getText());
    if (entry != null) {
        println(entry);
    }
}
```

Natürlich műssen wir vorher unsere Datenbank mit

```
db = new StockDataBase("SP500_HistoricalStockDataMonthly.csv");
```

initialisiert haben. Das müsste eigentlich genügen um die Funktionsfähigkeit unserer Datenbank unter Beweis zu stellen.

4. Grafik



Kommen wir zum Grafikteil. Im erste Schritt, machen wir aus dem *ConsoleProgram* ein *Program*. Wir könnten auch ein *GraphicsProgram* machen, das hat aber gewisse Nachteile wie wir gleich sehen werden.

Da ein *Program* noch keinen Canvas hat, müssen wir selbst für einen sorgen. Wir fügen also zu unserem Hauptprogramm die Instanzvariable *canvas* hinzu:

```
private StockCanvas canvas;
```

und initialisieren diese in der init() Methode:

```
canvas = new StockCanvas(db.getDates());
add(canvas, CENTER);
```

d.h. der Canvas bekommt eine Referenz auf die Dati und wir fügen den Canvas in den CENTER Bereich unseres Programms. Außerdem nehmen wir noch eine kleine Änderung an der *actionPerformed()* Methode vor:

```
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Graph")) {
        StockEntry entry = db.findEntry(tfSymbol.getText());
        if (entry != null) {
            canvas.addEntry(entry);
        }
    } else {
        canvas.clear();
    }
}
```

Wir sehen also drei Anforderungen an unsere *StockCanvas* Klasse: Der Konstruktor bekommt eine Liste mit Dati, und es muss die Methoden *addEntry()* und *clear()* geben.

```
public class StockCanvas extends GCanvas {
    private List<String> dates;
    private ArrayList<StockEntry> entries = new ArrayList<StockEntry>();

    public StockCanvas(List<String> dates) {
        this.dates = dates;
    }

    public void clear() {
        entries.clear();
        update();
    }

    public void addEntry(StockEntry entry) {
        entries.add(entry);
        update();
    }
}
```

Was jetzt noch fehlt damit alles hübsch wird, ist die *update()* Methode:

```
private void update() {
    removeAll();
    drawGrid();
    drawEntries();
}
```

Erst wird mal alles gelöscht, dann soll das Hintergrundraster mit den Beschriftungen gezeichnet werden, und darin sollen dann die Aktienkurse gezeichnet werden. Das sind aber "einfach" nur ein paar GLabels und GLines an der richtigen Stelle.

5. Resizing

"One more thing..." wie Steve Jobs zu sagen pflegte: was passiert eigentlich wenn wir die Größe unseres Fensters verändern? Ooops, das sieht aber hässlich aus. Was müssten wir tun, damit das Programm sich bei veränderter Größe neu zeichnet? Dafür gibt es das *ComponentListener* Interface: unsere *StockCanvas* Klasse muss dieses implementieren:

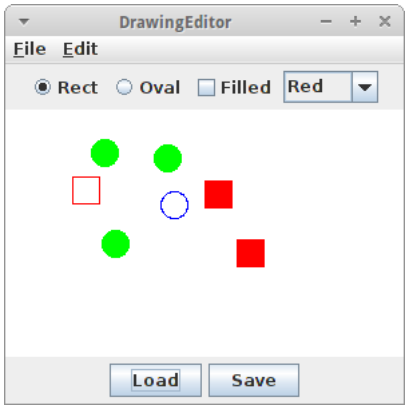
```
public class StockCanvas extends GCanvas implements ComponentListener {
    public StockCanvas(List<String> dates) {
        this.dates = dates;
        addComponentListener(this);
    }
    ...
}
```

im Konstruktor müssen wir uns selbst als *ComponentListener* hinzufügen, und es müssen zwingender weise die folgenden Methoden hinzugefügt werden:

```
    public void componentHidden(ComponentEvent e) {}
    public void componentMoved(ComponentEvent e) {}
    public void componentShown(ComponentEvent e) {}

    public void componentResized(ComponentEvent e) {
        update();
    }
```

die alle nichts tun außer die letzte. Und das genügt schon, damit unser Programm sich neu zeichnet wenn sich die Größe verändert.

DrawingEditor



Wir wollen ein einfaches Zeichenprogramm schreiben. Die Idee ist, dass wir zwischen den Formen Rechteck und Kreis mittels zweier Radiobuttons auswählen können, außerdem wollen wir festlegen können, ob die Formen ausgefüllt sein sollen oder nicht, und natürlich möchten wir noch die Farbe der Formen bestimmen können. Wir wollen aber nicht nur Zeichnen können, wir wollen unsere Kunstwerke auch Speichern und wieder Laden können.

Wir beginnen mit der UI. Dazu übernehmen wir einfach unsere Vorarbeiten aus dem Kapitel zu Swing, wo wir bereits die UI für einen solchen DrawingEditor geschrieben haben. Allerdings verschieben wir die Auswahl der Form in den Norden, und im Süden platzieren wir zwei JButtons zum Laden und Speichern.

Natürlich soll unser DrawingEditor ein GraphicsProgram sein, sonst wird das nichts mit dem Zeichnen. Da die Formen per Mausklick erzeugt werden sollen, benötigen wir einen MouseListener, und da wir auch auf die JButtons reagieren wollen ist zusätzlich ein ActionListener notwendig. Das alles machen wir in der *init()* Methode.

Da wir später alle unsere Formen (GObjects) speichern wollen, müssen wir sie natürlich im Auge behalten. Das heißt, wir benötigen eine ArrayList als Instanzvariable in die wir alle GObjects hinzufügen:

```
    private ArrayList<GObject> objects = new ArrayList<GObject>();
```

Arbeiten wir weiter an der *mousePressed()* Methode. Zunächst müssen wir wissen welche Form gewünscht ist. Wir fragen also die Radiobuttons:

```
GObject obj;
if (rBtnRect.isSelected()) {
    obj = new GRect(SIZE, SIZE);
} else {
    obj = new GOval(SIZE, SIZE);
}
```

Dann fragen wir die Combobox nach der Farbe,

```
Color[] colors = { Color.RED, Color.GREEN, Color.BLUE };
obj.setColor(colors[cBoxColorPicker.getSelectedIndex()]);
```

und schließlich sagt uns die Checkbox ob die Formen ausgefüllt sein sollen oder nicht:

```
GFillable flbl = (GFillable) obj;
flbl.setFilled(true);
```

Jetzt können wir die Form zum Canvas hinzufügen, aber wir sollten es zusätzlich in unsere Liste von GObjects einfügen:

```
objects.add(obj);
```

Damit ist der Zeichnen Teil geschafft.

Kommen wir zum Speichern unserer Kunstwerke. Bisher wenn wir Daten gespeichert haben, dann waren das immer Textdaten. Wie ist das aber mit binären Daten? Dafür gibt es die Klassen *FileInputStream* und *FileOutputStream*. Die erste zum Lesen, die zweite zum Schreiben. Das ist schon mal ne tolle Sache. Wer aber den Vogel abschießt, ist der *ObjectOutputStream*: der macht nämlich aus unseren Objekte serialisierte, binäre Daten. Wir müssen nur sagen *writeObject(objects)*, und den Rest erledigt der *ObjectOutputStream*:

```
private void saveFile(String fileName) {
    try {
        ObjectOutputStream oos =
            new ObjectOutputStream(new FileOutputStream(fileName));
        oos.writeObject(objects);
        oos.close();
        println("Save success.");

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Umgekehrt geht das beim Lesen ganz genau so einfach mir *readObject()*:

```
private void openFile(String fileName) {
    try {
        ObjectInputStream ois =
            new ObjectInputStream(new FileInputStream(fileName));
        objects = (ArrayList<GObject>)ois.readObject();
        ois.close();
        println("Load success.");

        // add objects to canvas:
        removeAll();
        for (GObject obj : objects) {
            add(obj);
        }
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
```

Da der *ObjectInputStream* ein bischen dumm ist, müssen wir noch einen Cast machen um wieder an unsere GObjects zu kommen. Und natürlich müssen wir die neuen GObjects noch zu unserem Canvas hinzufügen. Aber das war's dann wieder mal.

Library

Wir haben gesehen wie man mit Objekt-Orientierte Analyse aus Anforderungen (also Text) eine Programmstruktur mit verschiedenen Klassen erstellen kann. Genau das wollen wir jetzt mit dem Beispiel der Hochschulbibliothek machen:

"Die Hochschulbibliothek (Library) hat Studierende und Bücher. Eine Studierende hat einen Namen, eine Immatrikulationsnummer und eine Liste von Büchern die sie ausgeliehen hat. Ein Buch hat einen Autor und einen Titel. Eine Studierende kann Bücher ausleihen, sowie Bücher zurückgeben. Wir können alle Bücher auflisten lassen, die eine Studierende ausgeliehen hat. Wir können neue Studierende anlegen und wir können neue Bücher anlegen."

Es geht also darum die Klassen, Attribute und Methoden der Klassen zu bestimmen. Damit dies aber nicht nur eine theoretische Übung bleibt, wollen wir dann auch einen Schritt weitergehen, und das Programm wirklich schreiben. Hier macht es Sinn sich noch einmal kurz das Projekt *InteractiveMenuProgram* aus Kaptiel 3 anzusehen.

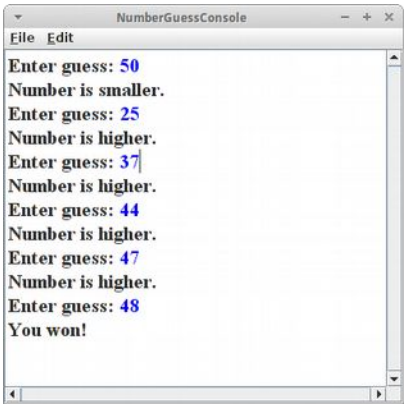
Mensa

Das Gleiche was wir für die Hochschulbibliothek getan haben, können wir natürlich auch für die Mensa tun:

"Die Mensa hat Gerichte und Zutaten. Ein Gericht hat einen Namen, einen Preis und eine Liste von Zutaten. Eine Zutat hat einen Namen, einen Preis und Kalorien. Es können neue Gerichte angelegt werden, sowie existierende Gerichte gelöscht werden. Wir können alle Zutaten auflisten, die zu einem Gericht gehören. Wir können neue Zutaten anlegen und wir können neue Gerichte anlegen."

Solange wir umsonst arbeiten, werden wir solche Aufträge im Stundenrhythmus erhalten.

NumberGuess



Bisher haben wir immer alleine programmiert. Was ist aber wenn mehr als eine Person an einem Programm schreiben sollen, wollen oder dürfen? Dafür gibt es Interfaces, also Schnittstellen.

Betrachten wir als Beispiel das Spiel NumberGuess. In dem Spiel geht es darum, dass der Computer eine Zufallszahl zwischen 0 und 99 wählt und der Spieler diese möglichst schnell erraten soll. Das Programm besteht aus zwei Teilen: dem Teil der sich um die Logik kümmert (*NumberGuessLogic*) und den Teil der mit dem Spieler interagiert (*NumberGuessConsole*). Wir haben also zwei Klassen, die allerdings beide voneinander abhängen. Diese Abhängigkeit wollen wir klar definieren, und das machen wir mittels einer Schnittstelle, also einem Interface:

```
public interface NumberGuessLogic {

    /**
     * guess should be number between 0 and 99 <br/>
     * return 0 if guess was correct <br/>
     * return +1 if guess was higher <br/>
     * return -1 if guess was lower
     */
    public int makeGuess(int guess);
}
```

Dieses Interface stellt einen Vertrag zwischen zwei Entwicklern dar.

- Für den einen (*NumberGuessConsole*) bedeutet es, dass es eine Methode *makeGuess(int guess)* geben wird, der ich die vom Spieler geratene Zahl übergeben kann, und die mir als Rückgabewert sagt, ob die Zahl zu klein, zu groß oder richtig war (-1, +1, 0).
- Für den anderen (*NumberGuessLogic*) bedeutet es, dass er eine Methode *makeGuess(int guess)* schreiben muss, die eine Zahl *guess* mit der intern generierten Zufallszahl vergleicht und dann Rückgabewert liefert der besagt ob die Zahl zu klein, zu groß oder richtig war (-1, +1, 0).

Dies erlaubt es zwei oder mehr Entwicklern unabhängig voneinander Code für ein Programm zu schreiben, der dann mühelos zusammengefügt werden kann.

Sehen wir uns das im Detail an. Wir beginnen mit dem Teil der das Interface *NumberGuessLogic* implementiert. Häufig nennt man Klassen die nichts anderes tun als ein Interface zu implementieren genauso wie das Interface, hängt aber noch ein "Impl" an den Ende des Namens:

```
public class NumberGuessLogicImpl implements NumberGuessLogic {
    // instance vars
    // constructor

    public int makeGuess(int guess) {
        ...
    }
}
```

Diese Klasse muss eine Methode namens *makeGuess(int guess)* haben, die jetzt natürlich auch etwas tun muss. Und natürlich muss irgendwo die Zufallszahl generiert werden, wahrscheinlich im Konstruktor.

Unabhängig davon kann der andere Entwickler bereits beginnen mit *NumberGuessConsole* Klasse. Dabei handelt es sich um ein ganz normales ConsoleProgram:

```
public class NumberGuessConsole extends ConsoleProgram {

    public void run() {
        NumberGuessLogic logic = new NumberGuessLogicImpl();
        ...
        if (logic.makeGuess(guess) == 0) {
            ...
        }
        ...
    }
}
```

In diesem Programm benutzen wir die *NumberGuessLogicImpl* Klasse und tun so wie wenn sie schon existieren würde. Wenn dann beide Entwickler fertig sind, fügen wir die Klassen einfach in ein Projekt zusammen, und alles sollte problemlos funktionieren.

Fragen

1. Schreiben Sie ein Methode countLinesInFile(String fileName), die als Rückgabewert eine Zahl (int) gibt, die die Anzahl der Zeilen in der Datei mit dem Namen „fileName“ beinhaltet. Benutzen Sie dazu die Klassen BufferedReader und FileReader. Sie müssen kein Exception-Handling (try-catch) machen.

2. Schreiben Sie ein Methode readFromFile(String fileName), die als Rückgabewert einen String zurückgibt, der den kompletten Inhalt der Datei mit dem Namen fileName beinhaltet. Benutzen Sie dazu die Klassen BufferedReader und FileReader. Sie müssen kein Exception-Handling (try-catch) machen.

3. Wenn wir von Dateien lesen wollen verwenden wir in der Regel den FileReader. Wenn wir unseren Code schreiben, dann zwingt uns der Compiler immer um den ganzen Code einen 'try-catch' Block zu machen. Warum?

4. Im folgenden Code kann es zu Exceptions kommen:
```
 BufferedReader rd = new BufferedReader(new
FileReader("students.txt"));
  while (true) {
      String line = rd.readLine();
      if ( line == null ) break;
      println( line );
  }
  rd.close();
```
Wie müsste der Code modifiziert werden, also was müsste hinzugefügt werden, damit die Exceptions abgefangen (catch) werden?

5. Was hat es mit dem "null" auf sich?

6. Erklären Sie den Unterschied zwischen einem Array und einer ArrayList.

7. Was ist der größte Nachteil eines Arrays gegenüber einer ArrayList?

8. Nennen Sie zwei Beispiele wofür eine HashMap besser geeignet ist als eine ArrayList.

9. Gibt es einen Unterschied zwischen den beiden Code Beispielen unten?

```
ArrayList<String> nameList = new ArrayList<String>();
for (String name : nameList) {
    println(" " + name);
}

ArrayList<String> nameList = new ArrayList<String>();
for (int i = 0; i < nameList.size(); i++) {
    String name = nameList.get(i);
    println(" " + name);
}
```

10. Hat ein Array auch Vorteile gegenüber einer ArrayList? Wenn ja, nennen Sie einen.

11. Nennen Sie zwei Beispiele wofür eine HashMap besser geeignet ist als eine ArrayList.

12. Geben Sie zwei Beispiele wofür man HashMaps gut gebrauchen kann.

13. In der folgenden HashMap soll ein Telefonbuch gespeichert werden:
```
private HashMap<String,Integer> phoneBook = new
HashMap<String,Integer>();
```
Geben Sie kurze Codeschnipsel, die zeigen
a) wie man ein neue Telefonnummer einfügt,
b) wie man die Telefonnummer einer Person zugreifen kann und
c) wie man alle Telefonnummern auflistet.

14. In der folgenden HashMap soll ein englisch-deutsches Wörterbuch gespeichert werden:
```
private HashMap<String,String> dictionary = new
HashMap<String,String>();
```
Geben Sie kurze Codeschnipsel, die zeigen
a) wie man ein neues Wortpaar einfügt,
b) wie man ein für ein gegebenes englisches Wort das entsprechende deutsche liest und
c) wie man alle Wörter auflistet.

15. Anforderungsanalyse (Requirements): In der Vorlesung haben wir die Anforderung des FlyTunesStore's analysiert. Benutzen Sie die selbe Technik um die Anforderungen des Journals für angewandte Informatik zu analysieren:

"Das Journal hat Artikel und Ausgaben. Ein Artikel hat einen Autor, einen Titel und einen Text. Eine Ausgabe hat einen Titel, eine Ausgabennummer und Artikel. Wir können alle Artikel auflisten lassen, alle Ausgaben auflisten lassen, sowie die Artikel für eine bestimmte Ausgabennummer. Wir können einen neuen Artikel anlegen und wir können neue Ausgaben anlegen."

Identifizieren Sie die notwendigen Klassen, Instanzvariablen und Methoden. Außerdem ordnen Sie die Instanzvariablen und Methoden der richtigen Klasse zu.

16. Gibt es für die Klasse GLabel die 'setFilled()' Methode?

17. Was ist der Unterschied zwischen einem 'interface' und einer 'class'?

18. Wenn wir Grafikobjekte, z.B. GOvals in einem Array speichern wollen, müssen wir bei der Deklaration den Datentyp angeben, also,
```
GOval[] circles = new GOval[4];
```
Das bedeutet aber, dass wir nur GOvals in diesem Array speichern dürfen. Wie müssten wir den Code ändern, so dass wir beliebige Objekte (also auch GLine, GRect, etc) in diesem Array speichern dürfen?

Referenzen

Auch das letzte Kapitel zehrt von den Referenzen des zweiten Kapitel. Auch ein sehr schönes Buch, das jetzt in Reichweite gerückt ist, ist das von Robert Sedgewick und Kevin Wayne [4].

[1] Design by contract, https://en.wikipedia.org/w/index.php?
title=Design_by_contract&oldid=700763992 (last visited Feb. 25, 2016).

[2] Ulysses, https://en.wikipedia.org/wiki/Ulysses_(novel)

[3] Latitude and longitude of cities, A-H,
https://en.wikipedia.org/wiki/Latitude_and_longitude_of_cities,_A-H

[4] Introduction to Programming in Java, von Robert Sedgewick und Kevin Wayne

[5] Real-World Data Sets, introcs.cs.princeton.edu/java/data/

[6] Stoppwort, https://de.wikipedia.org/wiki/Stoppwort

[7] QuantQuote Free Historical Stock Data, https://quantquote.com/historical-stock-data

Epilogue

Geschafft, meinen Glückwunsch! Ich hoffe es hat Spaß gemacht. Was kommt als nächstes?
Urlaub, Ferien, Pause. Punkt.

.

Index

A
ActionEvent 89, 91f., 130, 145, 147, 152f.
ActionListener 88ff., 92, 152, 154
Anforderungen 79, 136f., 153, 156, 159
Anforderungsanalyse 136, 159
Animationen 2, 45, 49ff.
Array 2, 80, 101ff., 109ff., 118, 124ff., 129f., 134ff., 141f., 144ff., 150ff., 158f.
ArrayList 2, 129, 134ff., 141, 144ff., 150ff., 158f.
Attributen 71, 138
AudioClip 109f.
Ausdruck 28f., 31, 33, 41f., 135

B
Bedingung 2f., 31ff., 75, 103, 118
Benutzeroberfläche 2, 87f., 94
Bildverarbeitung 2, 109
Boolesche 31ff., 37, 40, 42
BorderLayout 93
Buchstaben 11, 20, 27f., 30, 60, 67ff., 71, 75ff., 82f., 95, 104, 149
BufferedReader 132, 141, 143, 151, 158
Buttongroup 92, 97

C
Caesar 77, 85
Canvas 16, 18, 52f., 55, 89, 153ff.
Character 68f., 74, 76
Color 16ff., 21, 24, 46ff., 50, 52ff., 60, 65, 79, 92, 108, 112f., 117ff., 124, 126, 130, 134, 139f., 155
ComponentListener 154
ConsoleProgram 26, 30, 34, 38f., 41, 46ff., 72f., 77f., 85, 88ff., 92, 103, 151ff., 157

D
Datei 2, 24, 71, 97, 109, 131ff., 141ff., 146f., 149ff., 158
Datenstruktur 126, 131, 134f., 141, 147
Datentyp 27ff., 31, 34, 38ff., 46, 48, 52, 64, 68, 73, 96, 102, 135, 137, 159
Deklaration 27, 46, 51, 84, 103, 108, 123, 135, 159
Division 29f., 58, 112, 115, 126, 134, 142

E
ELIZA 79, 85
Elternklasse 50, 107f.
Endlosschleife 11, 36, 50, 55ff., 105f., 133
Ereignis 8, 52, 58, 99, 101, 105f., 109
Events 2, 51, 53, 61, 91, 95, 101, 105f., 120, 126, 128, 130
Exception 132ff., 151, 155f., 158

F
Fehlerbehandlung 2, 141
FileReader 132, 141, 143, 151, 158
FileWriter 132f., 141f., 144
Final 30, 85, 107, 113, 115, 123, 139
FlowLayout 93

G
GameLoop 53, 59ff., 63, 80, 82, 96, 105, 119ff., 125, 128f.
Ganzzahl 27ff., 96, 102, 112, 115, 126, 130, 142
Ganzzahldivision 29, 142
GArcs 22f., 56
GCanvas 53, 89, 153f.
GCompound 94, 101, 108f., 116, 140
GFillable 139f., 155
GImage 16, 18, 20, 49, 104, 111ff., 126, 140
GLabel 16, 19f., 49, 52, 56, 88, 115, 126, 139, 154, 159
Gleitkommazahlen 28f., 40
GLine 16, 18, 20, 27, 49, 55, 57f., 140, 154, 159
GObject 49f., 53, 59ff., 63f., 84, 108, 110, 124f., 129, 139f., 154ff.
GOval 16, 18, 20ff., 27, 47ff., 55f., 59, 61f., 65, 71, 84, 102f., 108, 119ff., 123ff., 127, 130, 139f., 143f., 155, 159
GPolygon 16, 20, 22ff., 49, 105ff., 120, 127, 129, 140
Grafikprogramme 15, 17, 25, 87
GRect 16ff., 20ff., 26f., 46ff., 52f., 55, 57, 60ff., 71, 84, 110, 116ff., 120f., 123ff., 127, 129, 134, 139f., 155, 159
GridLayout 93f., 97f., 113
Großbuchstaben 20, 30, 68f., 71, 75f., 104

H
HashMap 2, 134ff., 141, 145ff., 152, 158f.

I
Immutability 71
Inkrement 35, 37
Instanzvariable 2, 51f., 56ff., 72, 74f., 79, 82f., 87, 90ff., 94ff., 98f., 105, 107, 110, 120f., 123f., 143, 145, 151ff., 159
Integer 26f., 29, 58, 73, 112, 134f., 142f., 149, 159
Interface 2, 88, 123, 127, 131, 139ff., 154, 156f., 159
Iterationen 118

J
JButton 88ff., 93f., 96ff., 113, 145, 151, 154
JCheckBox 91
JComboBox 92
Jeliot 28f., 35, 43
JLabel 88ff., 95ff., 147, 151
JOptionPane 147
JPanel 94, 97
JRadioButton 92, 97
JTextArea 97, 144
JTextField 89f., 96ff., 117, 147, 151

K
KeyEvent 2, 61f., 95, 101, 105f., 120, 122, 128, 130
KeyListener 61f., 95, 105, 107, 120ff., 127
Kinderklassen 50, 64, 139
Kommentar 10f., 14

Komposition 2, 101, 105, 108f., 116, 140
Konsolenprogramm 2, 25f., 31, 38, 40, 67, 78, 142
Konstante 30, 37f., 42, 54, 99, 121ff., 127, 139
Konstruktor 72, 74, 79, 84, 90, 107f., 116f., 119, 144, 152ff., 157

L
Layout 93f., 97ff., 113

M
Mehrfachvererbung 108, 139
Modulo 30, 95
MouseEvent 2, 51f., 58f., 63, 105, 110, 115, 126f., 130
MouseListener 51f., 57ff., 63, 105, 110, 115, 126, 154

O
Objektorientierung 2, 20, 105, 108f., 116
Operationen 33, 37
Operator 2, 29f., 32f., 35, 37, 39f., 42f., 56, 77, 98, 113, 122

P
Parameter 39, 46, 48f., 54, 64, 72, 77ff., 82, 84, 90, 95, 111, 125, 139
Polymorphie 131, 141
Polymorphism 2, 140

R
RandomGenerator 2, 53, 55f., 64f., 75, 79, 90, 116, 142
Referenz 1, 3, 14, 24, 43, 65, 85, 99, 110f., 125, 130, 153, 160
Remainder 30, 43, 56, 122
Restwert 30, 37, 39, 77
Rückgabewert 46ff., 53, 64, 72, 76, 78, 82f., 90, 139, 157f.

S
Schaltjahr 33, 40
Schleife 2, 6ff., 11, 34ff., 43, 50, 55ff., 75, 81, 102, 104ff., 109, 111, 114, 133, 136
Schnittstelle 139, 156
Sentinel 36, 111, 141
Sichtbarkeit 46, 71
Sonderzeichen 28, 68
Split 78, 142ff., 149, 152
StringTokenizer 2, 70f., 74, 78, 84, 142, 146f., 149
Swing 1f., 87, 93f., 99, 144, 147, 154

T
Tastenereignisse 105f.
Typumwandlung 38, 75

U
Übergabeparameter 39, 48, 64, 111

V
Vererbung 2, 101, 105ff., 116, 119, 130, 139f.
Vergleiche 31, 42, 70, 76, 83f., 103, 130
Vorrangsregeln 33, 42

W
Wahrheitstabellen 33, 37, 40
Wiederverwendung 116, 119
Wrapper 73f.

Z
Zauberzahlen 38, 47
Zeichenketten 67f.
Zufallszahlen 53, 142
Zuweisung 27f., 34, 37f., 102

Č
Čapek 14