Variations on a Theme:
JavaScript
An object-oriented Introduction

by Ralph P. Lano, 1. Edition

Why

this book? I have always been writing little bits and pieces of JavaScript, but my preferred language always was Java, which is what I teach. What triggered this book, was a student with an iPad Pro, who asked if he could use it for my class. Unfortunately, the answer was no. Times change, and so did JavaScript. With the advent of ES6, some "syntactic sugar" was added to the language: classes. And since I really like sugar, I set myself the challenge: could I rewrite my introductory Java book in JavaScript? My main design criteria was to have as few changes as possible to convert Java examples to JavaScript. And surprisingly, it worked really well. This may not really be a good reason to write a book, but then, what is?

For whom

Well, obviously for the student with the iPad. Definitely, hardcore JavaScript folks will not like it, and this is NOT a mainstream JavaScript book. This is a fun project. And if you have little or no programming skills, maybe this book here will be a helpful introduction. If you like books with many pictures, then also, this book may be for you. But don't be fooled by the number of pages: there is a lot of work to be done. If a lot of work is not your thing, then this book probably is not for you.

From whom

I have been teaching Java for roughly twenty years now, in different forms, courses and universities. So naturally, this is JavaScript through the eyes of a Java developer. Since 2011, I have been Professor of Internet Programming and Multimedia Applications in the MediaEngineering program at the Nuremberg Institute of Technology. From 2003 to 2010 I was Professor of Software Engineering and Multimedia Applications at the University of Applied Sciences Hof, and from 2010 to 2011 Professor of Media and Computing at HTW Berlin. I received my doctorate in 1996 from the University of Iowa on the subject 'Quantum Gravity: Variations on a Theme'. From 1996 to 1997 I was Postdoctoral Research Associate at the Centre for Theoretical Studies of the Indian Institute of Science. From 1997 to 2003 I worked at Pearson Education and later at Siemens AG in software development and project management.

About what

It's getting a bit technical now, but some people want to know what they're getting into: In the first chapter we take the first programming steps with Karel. We get to know functions, loops, conditions and our first software engineering principles. Above all, the top-down principle is introduced for the first time. The second chapter introduces graphics programming, we learn to use objects using a JavaScript version of the ACM graphics library. This is followed by an introduction to console programs. The point here is to get to know variables and operators, conditions and loops are deepened, boolean variables are introduced and the 'Loop and a Half' is shown. In the chapter Agrar the top-down principle is deepened and functions are put on a new foundation. It is also shown how animations and mouse events work, and the random generator is used for the first time. Chapter five deals with strings and introduces classes. The Swing chapter introduces the basics of graphical user interfaces and concretizes the concept of instance variables. The basics of object orientation, inheritance and composition, will be discussed in detail in the following chapter and will be deepened by numerous examples. Arrays and simple image processing are also introduced. Then we finish off the main part with the introduction of the list and map data structures, and we conclude with interfaces and loosely introduce the term polymorphism.

The appendix starts with a primer on JavaScript and some of its new features. This is followed by a brief description of each of the libraries we use in the main part. Especially the graphics libraries are nothing but a thin wrapper around the p5.js library from the Processing Foundation [7]. The final chapter, is just a loose collection of idea, which is work in progress.

How

does one learn to play the piano? Not by watching or listening. It's the same with programming: you have to practice a lot! And the more you practice, the better you become with programming. We're not only learning

programming here, but also actually taking our first steps in software engineering. But as Johann Sebastian Bach so nicely said: "All you have to do is hit the right key at the right time."

The course as I teach it, consists of three components: the lecture, the labs and homework. The lecture is four hours per week and corresponds to the first part of each chapter in the book. We complete one chapter in about one to two weeks. In the labs, which also last four hours a week, we then devote ourselves to the projects. We manage between four and six of the projects per lab. In the labs, the students work in teams, usually in pairs, to help each other.

Homework is done on a weekly basis and takes about 8 to 10 hours per week. It is important that the students work alone on their homework. I attach great importance to the fact that the students commit themselves to follow the Stanford University honor code. The homework also doesn't have to be complete or work perfectly. The homework is discussed individually in the labs, and almost all problems related to the homework usually can be solved within five minutes.

Where

can I find the examples and the source code? You can find them on the website of the book: https://github.com/rplano. Also, the book in electronic form, including updates, can be found there. The book in printed form is available on Amazon, in black and white.

May I

use the examples, or copy the book? This material is licensed under Creative Commons Attribution - Non-commercial - Distribution under Equal Terms 4.0 International (CC-BY-NC-SA 4.0). That is, you may reproduce and redistribute the material in any format or medium, remix the material, modify it, and build upon it. However, you must provide appropriate copyright and other proprietary notices, include a link to the license, and indicate whether any changes have been made. This information may be provided in any reasonable manner, but not in such a way as to create the impression that the Licensor is specifically endorsing you or your use. You may not use the material for commercial purposes. And if you remix, modify, or otherwise directly build upon the material, you may distribute your submissions only under the same license as the original, and you may not use any additional clauses or technical procedures that legally prohibit others from doing anything that the license permits. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/.

The source code is available under the MIT License (http://choosealicense.com/licenses/mit/).

From where

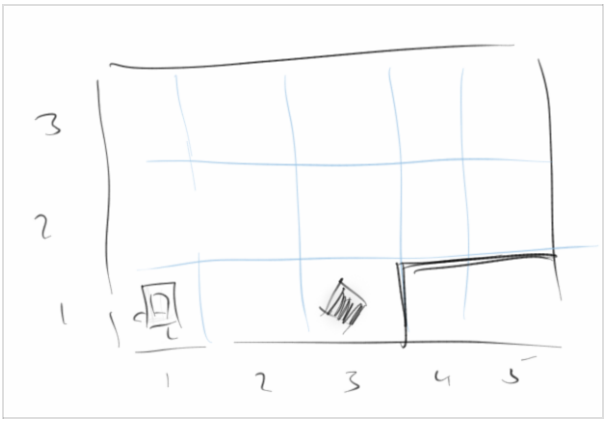
come the ideas? Since this book is almost identical to my first book, the style, the pedagogical structure and many of the examples are heavily inspired by Mehran Sahami's lecture [1], which in turn is based on the book by Eric Roberts [2]. I personally learned JavaScript from David Flanagan's book [5]. However, JavaScript has changed quite a bit over the years, and if you really want to understand it, you must read Douglas Crockford's book [3]. The tutorials on W3Schools [4] are really helpful when you need a quick solution for a simple problem. Addy Osmani website [6] I really like, because it shows that you can write really nice, clean code with JavaScript, just take a look for yourself.

References

[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] JavaScript: The Good Parts: Working with the Shallow Grain of JavaScript, Douglas Crockford, O'Reilly, 2008
[4] JavaScript Tutorial - W3Schools, www.w3schools.com/js/
[5] JavaScript: The Definitive Guide, David Flanagan, O'Reilly, 2012
[6] Learning JavaScript Design Patterns, Addy Osmani, https://www.patterns.dev/posts/classic-design-patterns/
[7] p5.js, https://p5js.org/

.

.



KAREL

Karel is a little robot that looks a bit like an old iMac. On the following pages we will learn a lot from Karel. Although it seems trivial, in this chapter we probably learn the most important lesson of the whole book, namely to break down complex problems into simpler ones with the top-down approach.

A little note for people who already know how to program a bit: don't skip this chapter! In principle, this chapter is about learning and practicing the top-down approach with simple examples, getting used to a good coding style, and learn how to code without using variables.

.

Karel's World

Karel's world consists of streets and avenues. Streets run from west to east and avenues from south to north. Besides, there are walls Karel can't go through, and there are beepers. Karel always carries a bag with an infinite number of beepers, but beepers can also lie around anywhere in his world.

Karel knows only four commands:

- **move():** he can move one step forward,
- **turnLeft():** he can turn to the left,
- **pickBeeper():** he can pick up a beeper, and
- **putBeeper():** he can put a beeper down.

Although it doesn't look like much, it turns out that Karel can calculate anything there is to calculate, we say Karel is a 'universal computing machine'.

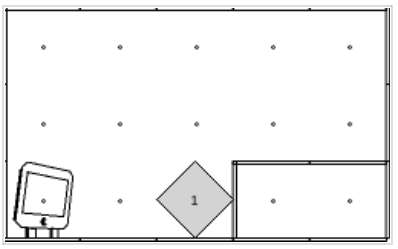
.

Karel learns to walk

If we want Karel to pick up the beeper at location (3,2) and carry it to location (5,2), we would tell him:

"Karel, take two steps forward, then pick up the beeper, turn left, move one step forward, turn three times to the left, move two more steps forward and put down the beeper."

Why should Karel turn three times to the left? Because he doesn't know how to 'turn to the right', nobody has ever shown him that.



Try it

Since Karel doesn't understand English, we have to talk to him in his language. which sounds like this:

```
move();
move();
pickBeeper();
```

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

Like nobody really understands why you need these funny punctuation marks in English (like commas, periods etc.), nobody knows why Karel needs these round brackets (called parentheses) and semicolons. But we can't do without them. Karel also pays meticulous attention to upper and lower case, if we make a typo he does nothing at all.

.

Karel Program

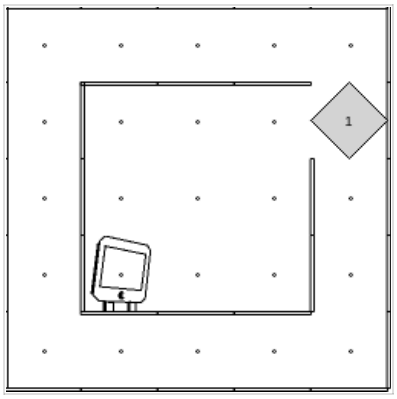
So for Karel to start his chores we need something called a 'program'. This is actually JavaScript, but we don't need to know that. A Karel program looks like this:

```
function run() {
    move();
    move();
    pickBeeper();
    turnLeft();
    move();
    turnLeft();
    turnLeft();
    turnLeft();
    move();
    putBeeper();
}
```

The part that is important for us is marked in blue. In all that follows we will simply insert our Karel commands (also called code) replacing the blue lines above. We don't care about the rest of the program.

.

Exercise: GoodMorningKarel



Try it

We want to solve our next Karel problem: Karel has just woken up and wants to drink his morning milk. However, the milk is still at the doorstep where the milkman left it. So Karel has to get up, go to the door, get the milk and then sit down at his breakfast table to drink his milk. How would that look in Karel's language?

.

Karel learns new Tricks

Karel is a bit like a pet and we can teach him new tricks. First, let's teach him to turn to the right. We tell him that every time he sees the turnRight() command, he should turn left three times. For Karel to understand us, we have to use his language:

```
function turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
}
```

Now we can just use this new command and Karel understands what to do. Commands we sometimes also call

functions, e.g. turnRight() is a function, but also move(). In general, functions are always indicated by parentheses.

Question: What should be the syntax for a turnAround() command that causes Karel to turn around and look in the opposite direction?

.

Exercise: GoodMorningKarel

To see that this really works, we want to modify our GoodMorningKarel so that it uses a turnRight() instead of three turnLeft()'s.

.

Karel repeats himself

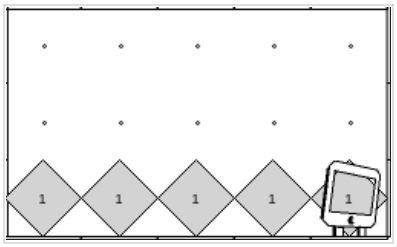
It is very common that Karel should do something over and over again. For example, he should turn three times to the left or he should walk two steps. For this we have something called a loop, more precisely a 'for loop'. In Karel's language this looks like:

```
function turnRight() {
    for (let i = 0; i < 3; i++) {
        turnLeft();
    }
}
```

What is important here is the '3', which tells Karel that he should do whatever is inside the curly brackets, three times. We do not need to understand the rest for the time being. (Very few people understand it, they just copy the lines and change the '3'.)

.

Exercise: FillRowKarel

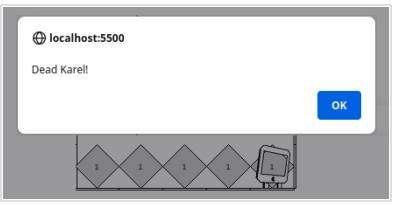


Try it

Another application for our loop is FillRowKarel. We want Karel to put five beepers in a row. This actually seems to be a very simple problem, and one would think that the following lines

```
function run() {
    for (let i = 0; i < 5; i++) {
        putBeeper();
        move();
    }
}
```

should solve the problem.



However, we realize that Karel runs into the wall at the end and hurts himself. In the program we see that when the red window pops up in which Karel is stung by a wasp. To avoid this, we could only go through the loop four times. But then Karel would only put down four beepers and not five. This problem occurs so often that it has a name: it is called the O-Bob problem.

.

.

OBOB

O-Bob was a Jedi master who first noticed this problem when he tried to program his droid. Seriously, OBOB actually stands for 'off by one bug', usually one to few. In our FillRowKarel problem this means that Karel simply has to put an extra beeper in the end:

```
function run() {
    for (let i = 0; i < 4; i++) {
        putBeeper();
        move();
    }
        putBeeper();
}
```

Right now, we have to live with that. In the next chapter we will see how the 'Loop-and-a-half' can be used to avoid the O-Bob problem elegantly.

.

Karel has Sensors

There's one little thing I haven't told you yet: Karel has sensors. For example, he can sense if there is a beeper at the location he is at the moment. So if we wanted Karel to pick up a beeper, if there is one, we could use the following Karel commands:

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

So Karel has another command, the *if* command.

Karel has a lot of sensors, the most important ones are:

- **beepersPresent():** there is a beeper at the current location,
- **noBeepersPresent():** there is no beeper at the current location,
- **frontIsClear():** there is no wall in front of Karel, and
- **frontIsBlocked():** there is a wall in front of Karel, so Karel should not try to move forward or he will hit his head.

Additionally, there are also sensors for rightIsClear() and leftIsClear(), together with their respective blocked variants. Right actually means down, and left actually means up, always relative to the direction in which Karel is looking.

.

Karel, go ahead

Very often we want Karel to do something until something happens. For example, as long as there is no wall in front of him, Karel should move straight. This is where Karel's while command comes in handy:

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

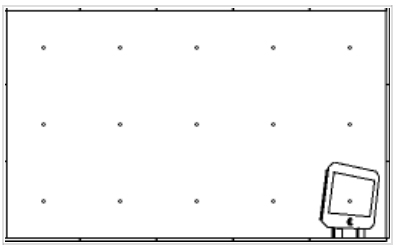
i.e., as long as there is no wall in front of him, Karel should move straight. This construct is called the 'while loop'.

.

Exercise: WallKarel

How would a new command *moveToWall()* look like, which causes Karel to move to the next wall? In WallKarel we want to get Karel to walk straight until a wall comes up. We want to use our new command moveToWall(), because then the code becomes very simple and elegant.

This example leads us to our first Software Engineering Principles (SEP):



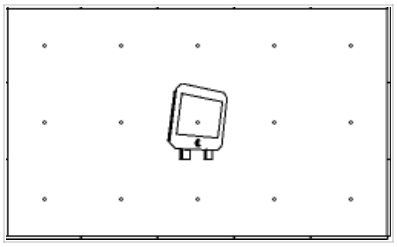
Try it

SEP: Programs should be readable by people and should look like relatively normal English.

.

.

Exercise: InfiniteLoopKarel



Try it

The while loop is not entirely unproblematic: because Karel does something until something special happens. In our InfiniteLoopKarel example, Karel should turn to the left until there is no more wall in front of him:

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

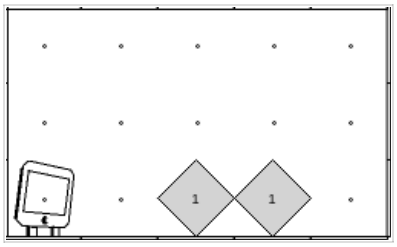
This works well as long as Karel is somewhere next to a wall. But if Karel is in the middle of his world, he does not stop turning. This is then called an infinite loop or Merry-Go-Round Karel.

.

.

.

Exercise: RobinHoodKarel



Try it

Karel loves the movies. The other day he saw the movie 'Robin Hood'. According to his motto 'take from the rich and give to the poor', Karel walks through his world and every time there is a beeper in the streets he picks it up, and whenever there is no beeper, he puts one down.
To do this, we can use Karel's sensors:

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

If we put these lines inside our while loop above, then we almost solved our problem. It works fine if there is no beeper. However, if there is one, Karel first removes it. But then he sees that there is no beeper, and hence puts a beeper where he shouldn't. To solve this problem we need the else command:

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

It reads like this: if there's a beeper pick it up, otherwise put one down.

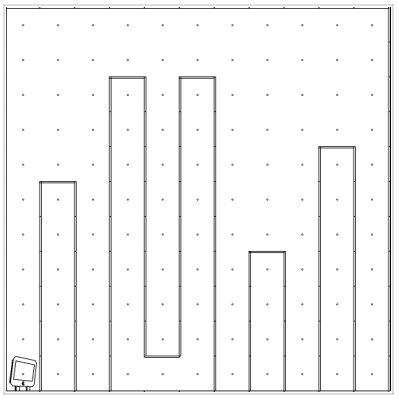
.

Top-Down Approach

In principle, there are two ways to solve any problem. One is the bottom-up approach: starting from the things you know, you try to solve the problem. So far we have used this approach to solve our Karel problems: with the few commands Karel knows and his sensors, we have solved simple problem step by step. For simple problems this works quite well.

For more complex problems, however, this no longer works very well. The top-down method becomes extremely helpful here. In the top-down approach we usually have a problem, which seems impossible to solve at first. However, one very often succeeds in breaking down the complex problem into smaller sub-problems. Sometimes those can be solved directly, sometimes those have to be broken down into even smaller sub-problems. This process is called 'stepwise refinement'. Let's take a look at the WindowCleaningKarel example, and see how it works.

Exercise: WindowCleaningKarel



Try it

Karel lives in Chicago. And he makes a living by cleaning windows. The job is not entirely harmless, since he is cleaning skyscrapers. Karel has to clean five skyscrapers a week. They are not all the same, some are higher some are less high. Actually, Karel likes the higher ones better, but they're more work.
We want to solve this problem using the top-down approach. It is therefore a question of breaking down the big problem into smaller ones. Suppose we knew how Karel cleans one skyscraper, so assume we had a function called *cleanOneSkyScraper()*. Then the solution to our problem would be very simple, we would just use cleanOneSkyScraper() five times:

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

So we've solved our big problem.
Now we have to solve the slightly smaller problem cleanOneSkyScraper(). How does Karel clean one skyscraper? Still the problem is to complicated to solve it with move's and turnLeft's. Hence, we need to break it down one more time. If we knew how Karel moves up the skyscraper, moveUpAndClean(), moves over the skyscraper, moveOver(), and on the other side then moves down the skyscraper again, moveDownAndClean(), then we would have solved the problem cleanOneSkyScraper():

```
function cleanOneSkyscraper() {
    moveUpAndClean();
    moveOver();
    moveDownAndClean();
}
```

Although we are not quite finished yet, we can see that we are basically done. We can easily solve the last three commands. For moveUpAndClean(), we let Karel turn to the left, and then let him move as long as his right side is blocked. For moveOver(), we let Karel turn right, move two steps forward and then turn right again. moveDownAndClean() is actually like moveToWall(), but at the end he has to turn left once more.

.

Recommendations for the Top-Down Approach

Sometimes it is not quite clear when the top-down approach is finished. Here are a few recommendations to help you:

- one function should solve exactly one problem,
- a function should not have more than 15 lines of code, around five lines is ideal, and
- function names should describe what the function does.

.

Comments

Every good program has comments. The more important it is, the more comments it has. What are comments? Comments are descriptions of what the program, what the code does. They are intended for people who are trying to understand the program. The computer ignores the comments.
The syntax is actually quite simple: a comment starts with the characters '/**' and ends with the characters '*/'.

Usually a comment is located at the very beginning of a program and explains what the program as a whole does. In addition, you should briefly describe what each function does. If you want to do it really well, then you also describe briefly which assumptions you make when the function is called (pre-conditions) and in what state you leave the world after the function is finished (post-conditions).

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

/**
 * Karel has to clean one skyscraper
 *
 * PreCondition: Karel is standing in front of the one skyscraper, facing
 * east <br/>
 * PostCondition: Karel is standing behind this one skyscraper, facing east
 */
function cleanOneSkyscraper() {
    moveUpAndClean();
    moveOver();
    moveDownAndClean();
}
...
```

If the functions are kept short, comments within functions are not necessary. Again, the point is that programs should read like good English.

.

Exercise: Comments

As a little exercise, we want to add detailed comments to the WindowCleaningKarel program. For all future programs we write, we should get used to adding a little comment to each program and function. This makes the programs easier to understand (for people).

.

REVIEW

In this chapter we have taken our first programming steps by helping Karel to solve his problems. In addition,

- we have taught Karel new commands, also called functions, e.g. moveToWall(),
- we let Karel repeat things with the for loop, e.g. make three steps forward,
- we have seen that Karel has sensors and we got to know the if statement,
- we learned about the while loop,
- we acquainted ourselves with our first software engineering principles,
- we made our first programming errors, e.g. the OBOB and the infinite loop,
- and we learned about comments.

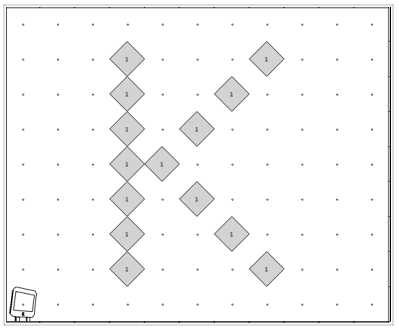
But the most important thing we learned in this chapter was the top-down approach. Its goal is to break down a big problem into smaller ones, and to repeat this until each sub-problem is easy to solve. This approach can be applied not only to programming problems, but also to solve problems in all possible other fields.

PROJECTS

We know enough about Karel now to help him solve any problems he may have. In the book by Pavel Solin [4] there are many more Karel examples.

.

WritingKarel



Karel learned to write in school. His favorite letter is the 'K'. So we want to help Karel write his favorite letter using the top-down approach.

.

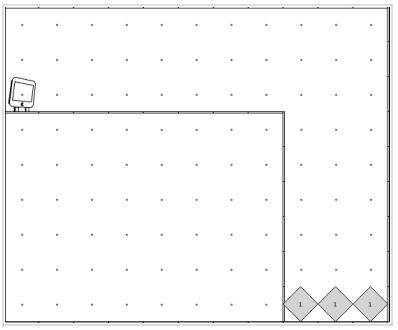
.

.

.

Try it

AdventureKarel



Karel is not a coward, on the contrary, he loves adventures. But that can be quite dangerous sometimes. The other day he was at the White Cliffs of Dover. Of course he wants to get as close as possible, but he doesn't want to fall over either (Karel can't swim). What makes this problem hard is that we don't know how far the cliffs are.

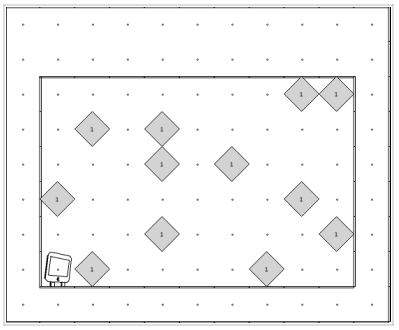
.

.

.

Try it

PartyKarel



Karel lives in a flat, which he shares with his brother SuperKarel. Both love parties and after the last one it looks like cabbage and turnips in their apartment. Tonight Karel's parents are coming to visit, so he has to clean up. Let's help Karel fix his apartment by cleaning up all the beepers lying around.
The part that makes this program a bit complicated is that Karel should stop after cleaning everything up.

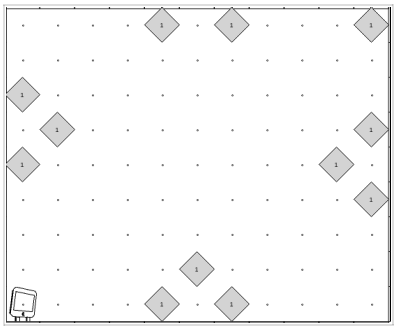
.

.

.

Try it

EasterEggKarel



Karel loves Easter and searching for Easter eggs. His parents were hiding them all over the garden, so we'll help Karel collect all the Easter eggs. Since he will visits his grandparents later, and their garden is much larger, the program must also work for gardens of different sizes.
Again, we want to make sure Karel stops when he is finished.

.

.

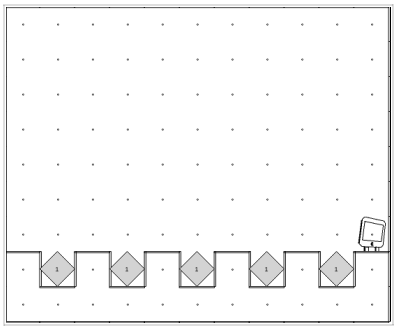
.

Try it

.

TulipKarel

Karel has Dutch ancestors, that's why he loves tulips. And every winter he plants tulip bulbs, so that his garden is full of tulips again in spring. Let's help Karel with planting the tulips, but using the top-down approach. Since Karel also plants tulips for his parents and grandparents, our program has to work for differently sized gardens.



Try it

.

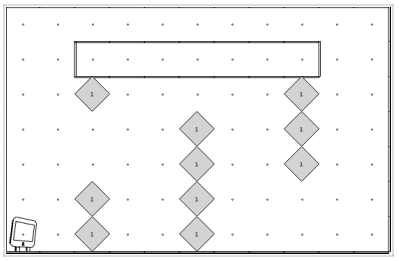
.

.

.

BuilderKarel

From time to time there are tornadoes in Chicago. The last one got Karel's house. It is still standing but the pillars have gotten some damage and have to be repaired. In Karel's house there are three pillars, which are of course made of beepers. The three pillars are three steps apart, but could be of different heights. Let's help Karel repair his house, of course with the top-down approach.



Try it

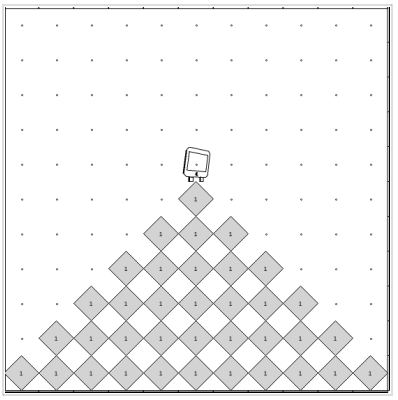
.

.

.

PyramidKarel

We have already heard that Karel is a little adventurer. The other day he saw a report on TV about the pyramids, and of course he has to build one in his garden immediately. Since he wants to build several, of different sizes, our program should work in worlds with different sizes as well, so it makes sense to use the top-down approach again.



Try it

.

.

.

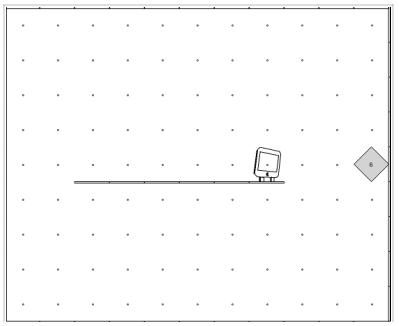
.

.

.

YardstickKarel

Karel is not quite as stupid as he looks. He can actually count even though he has no fingers (or variables). Karel counts with beepers. In this example he is supposed to measure how long the yardstick in front of him is. So we want him to put as many beepers as the yardstick is long, at the end of the street he is standing on.
We can assume that Karel is standing just before the yardstick.



.

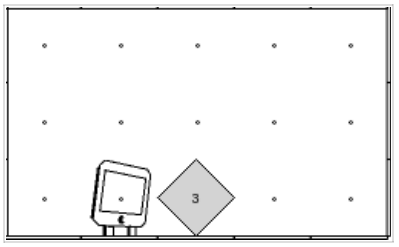
.

.

Try it

.

DoubleBeeperKarel



Karel can also do math. In this example he should double the number of beepers lying on the pile in front of him. After being finished, we want Karel to stand in front of the pile again. Very important: Karel does not know about variables, and although he can count (see last example) this does not really help. Furthermore, the program should work for any number of beepers.

The easiest way to solve this problem is the top-down approach. How about turning one beeper into two? And you do that until there are no more beepers.

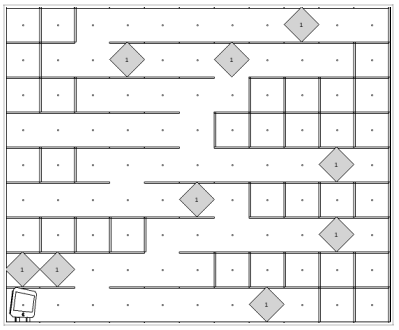
Try it

Question: What would the code look like if Karel is to triple or halve the number of beepers?

So Karel can do any of the basic arithmetic operations, and with those he can actually calculate anything there is to be calculated, one also says Karel is a *Universal Computing Machine*.

.

DonkeyKongKarel



Karel likes to play computer games when not cleaning windows. Preferably the classic arcade games of the 80's. His favorite game is Donkey Kong. This game is about collecting as many beepers as possible and moving from one level to the next. The exit is at the top right and we can assume that one level consists of nine floors.

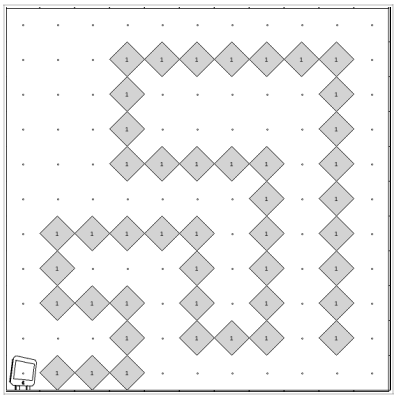
Of course we use the top-down approach again.

.

.

Try it

BreadcrumbKarel



Karel read about the Cretan princess Ariadne and how she helped Theseus to escape the labyrinth of the Minotaur. So he thought he could do the same thing using breadcrumbs.

.

.

.

Try it

QUESTIONS

1. Name the four most important commands of Karel.

2. Give an example of an Off By One Bug (OBOB).

3. In the exercise "PartyKarel" you taught Karel how to clean up. Explain briefly how Karel will clean up the beeper chaos. (No code required, just outline your procedure in words)

4. Like 3) only with PyramidKarel, DoubleBeepersKarel, WindowCleaningKarel,...

5. In the lecture we got to know recommendations for the top-down approach. These are rules regarding the names of functions, how many lines of code a function should have, etc. Name two of these guidelines.

6. What are comments good for?

7. What is the difference between bottom-up design and top-down design? Which is preferable?

8. Karel watched the film 'Robin Hood' and was totally impressed by the hero. That is why he wants to follow his example "take from the rich and give to the poor". So you should write a Karel program in which Karel picks up a beeper when he finds one and puts one down when there is none. To solve the problem, you may make the following assumptions about the world:
   • The world is at least 3x3.
   • At the start, Karel stands at the corner of 1st Street and 1st Avenue, looking east and has an infinite number of beepers in his beeper bag.
   Make sure that you only use Karel commands.

.

REFERENCES

The idea behind Karel comes from Rich Pattis, a former student of Stanford University [1]. The name 'Karel' is inspired by the first name of the Czech writer Karel Capek in whose play R.U.R. (Rossum's Universal Robots) the word 'robot' appears for the first time [2]. More details about Karel with many examples can be found in the Karel Reader [3]. Many of the examples used here are from the Karel Reader [3] and the Stanford Lecture 'Programming Methodologies' [4]. Even more Karel examples can be found in the book by Pavel Solin [5].

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

[2] Seite „Karel Capek“. In: Wikipedia, Die freie Enzyklop�die. 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

Writing graphics programs is almost as easy as writing Karel programs. In this chapter we will learn how to write our first graphics programs. We are also taking our first steps towards object-oriented programming.

.

Graphics Model

The graphic model we will use is very simple and may remind us of our time in kindergarten, when we still created the greatest works of art with pieces of felt. There are different pre-cut shapes, but you can also cut out your own shapes and arrange them on a felt board to create major works of art.

In our 'felt assortment' there are rectangles (GRect), circles (GOval), lines (GLine), polygons (GPolygon), arcs (GArc), images (GImage) and labels (GLabel). These can be of any color and size, and we can place them anywhere on our felt board.

.

Graphics Programs

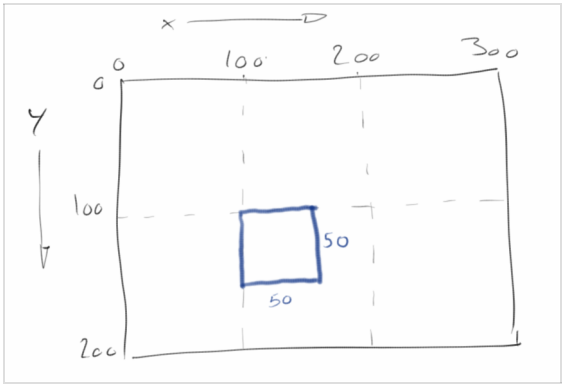
Our basic structure, so to speak our felt board, is the graphics program. It is very similar to our Karel programs:

```
function setup() {
  createCanvas(300, 150);
  frameRate(5);
        // your code...
}

function draw() {
  update();
}
```

Instead of the *run()*, we now have the *setup()* function. So our code goes into the *setup()* function We will ignore the *draw()* function for quite a while.

.

GRect



We get started by drawing rectangles, which we call GRect. A rectangle has a width and height, as well as an x and y position. The following two lines show how to create and draw a rectangle:

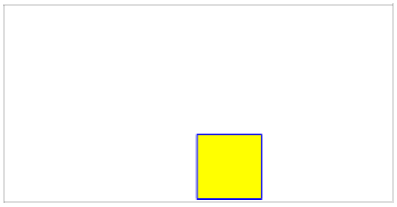
```
let fritz = new GRect(50, 50);
add(fritz, 150, 100);
```

In the first line we create a new rectangle that is 50 pixels wide and 50 pixels high. The rectangle also gets a name, it is called 'fritz'. The name can be almost arbitrary, and 'fritz' is as good a name as any. It's like we picked one of the felt rectangles and we are still holding it in our hands. Next, we have to add it to our felt board, and we do that with the second line: we add 'fritz' at the position (150, 100) to our felt board.

The felt board is actually called canvas. It is usually a little more than 300 pixels wide and just under 150 pixels high. But we can also make it bigger or smaller. Very important, x goes from left to right as usual, but y goes from top to bottom, this may be a little unusual.

.

Colors



Try it

Playing with felt pieces would be quite boring if all felt pieces had the same color. It is the same with graphics programs. So if we want to change the color of our rectangle, we set its color with the following line:

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

In this case, the border of 'fritz' is drawn in blue. In fact, we say, 'hey fritz, why don't you change your color to blue'.

Filled rectangles are of course much nicer. So we can tell 'fritz' that it should be filled:

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

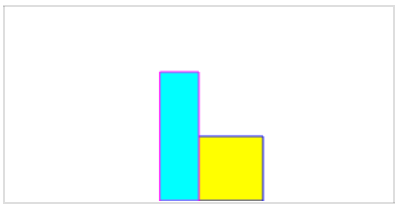
Sometimes you want the color of the border to be different from the color of the inside. For this 'fritz' has the setFillColor() method:

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

Now 'fritz' is a yellow rectangle with a blue border.

.

Multiple Rectangles



Playing with felt pieces would also be boring if there were only one felt piece. Similarly, we can add as many rectangles as we want. We just have to make sure that every new rectangle has a different name:

```
let lisa = new GRect(30, 100);
lisa.setColor(Color.MAGENTA);
lisa.setFilled(true);
lisa.setFillColor(Color.CYAN);
add(lisa, 120, 50);
```

If we add these lines to our program above, we see two rectangles, 'fritz' and 'lisa'.

.

Messages

Messages, functions, methods, commands: different names for the same concept. With Karel they were called move() and turnLeft(). Now they are called setColor() and setFilled(). We can write our own, like moveToWall(), or use existing ones. With Karel it was a little easier, because there was only one Karel. Now it gets a little bit more complicated, because there can be more than just one rectangle. That's why we always have to say for whom the message is intended, so

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

changes the color of 'fritz', but

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

changes the color of 'lisa'. In the first case we send the message to 'fritz', in the second to 'lisa'. What message are we sending? The 'setColor()' message. Alternatively, we say that we call the 'setColor()' method of 'fritz', or we let 'fritz' execute the 'setColor()' method.

.

Exercise: Flag

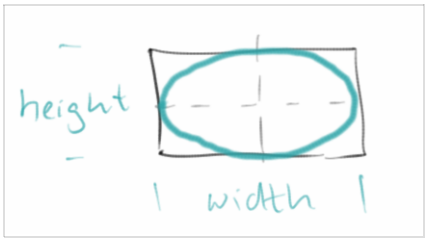
Karel always wanted to travel. To get him in the mood for his big trip, we want to write a graphics program that draws the flag of the country he would like to travel to. (However, Karel only likes countries with simple flags!)

.

Try it

.

GOval



If we want to draw circles or ellipses then we use GOval. GOval has a width and height like a rectangle, and the ellipse drawn is bordered by this rectangle, so to speak. Otherwise a GOval has the same methods as a GRect. The syntax is also identical:

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

.

GLine

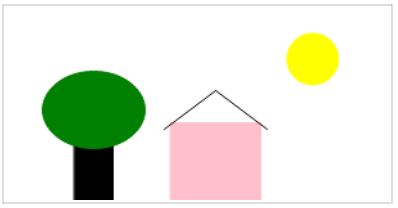
To draw lines there is the class GLine. It needs the x and y coordinates of the two points that should be connected by the line:

```
let leftRoof = new GLine(125, 95, 165, 65);
add(leftRoof);
```

Lines can also be colored. If you want to, you can also change the thickness of lines, or you could draw several lines next to each other.

.

Exercise: House with Tree



Try it

By now we know enough to create our first piece of digital art: as in kindergarten we want to draw a tree, a house with a roof and the sun. For this we use nicely colored GOvals, GRects and GLines.

.

.

.

GImage

Try it

Quite useful is the GImage class. It allows us to add images to our canvas:

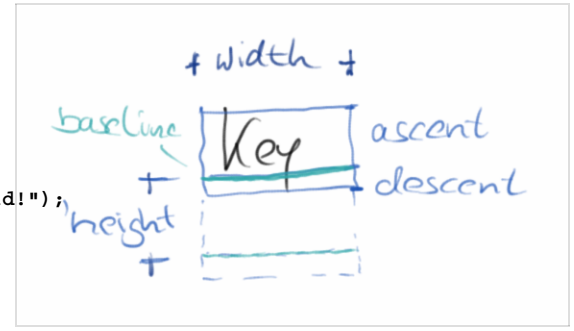
```
let om = new GImage(60, 10,
         "Ch2_Graphics/TH-Nuernberg-Logo.jpg");
om.scale(1.7);
add(om);
```

The GImage class understands the gif, jpg and png formats. Also, images can be scaled with the *scale()* method.

.

.

GLabel



From time to time we need some text with our graphics, then the GLabel comes in handy. It is as easy to use as the other graphic objects:

```
let hans = new GLabel("Hello World!");
add(hans, 40, 85);
```

If we don't like the font of our label, we can change it. You can choose the font type, the font size, and if you like you can also use an italic or bold font:

```
hans.setFont('Arial');
hans.setFontSize(36);
hans.setColor(Color.RED);
```

You have to be a little careful with the positioning: GLabels are positioned with respect to their baseline.

.

Exercise: HelloWorld

Try it

As a little exercise we write our first 'Hello World' program. Simply insert the lines above into the *setup()* function of a graphics program and execute it.

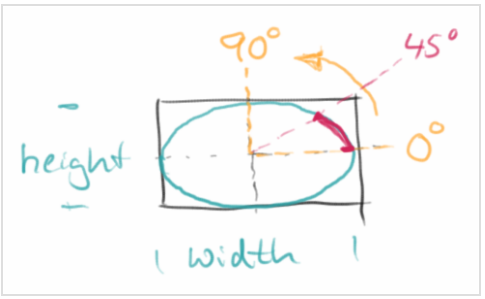
.

.

.

.

GArc



GArc draws an arc, i.e. a part of a circle or an ellipse. As with the ellipse we have to specify its width and height, as well as where the arc should be drawn. In addition, we need to specify which part of the arc should be drawn. This we do by specifying where the arc should begin (in degrees) and how long it should go (also in degrees):

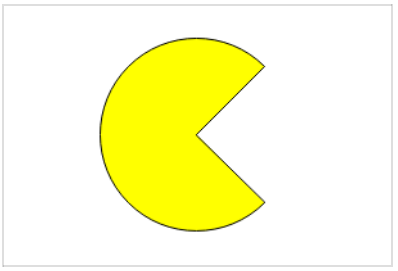
```
let archie = new GArc(150, 150, 0, 45);
add(archie, 75, 25);
```

The above lines draw an arc starting at 0 degrees and spanning 45 degrees counter-clockwise.

In the projects we will see how useful and versatile arcs are.

.

Exercise: PacMan



A first practical application for an arc is PacMan. It is simply a filled GArc starting at 45 degrees and spanning an arc of 270 degrees.

.

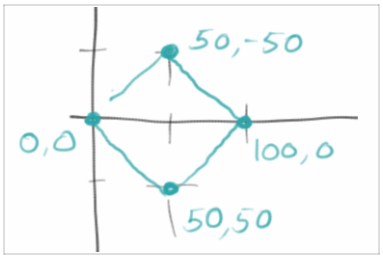
.

.

.

Try it

GPolygon



GPolygon is the last graphics class we need to learn. A polygon is made up of points, also called vertices, connected by lines. With addVertex() we add one point after the other. The polygon itself knows how to connect the points. Polygons are always closed, so you can fill them if you want to.

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

.

Window Size

Sometimes we want a different sized window. This can be done quite simply by modifying the line

```
createCanvas(300, 150);
```

at the beginning of our *setup()* method.

.

Object Orientation

Although we have not explicitly mentioned this, in the examples above we have been using objects and classes. Because fritz, lisa, hans and archie are objects, and GRect, GOval, GLabel are classes. At the beginning it is not always easy to distinguish between the two, but it actually is not that difficult:

- For example, we say 'fritz is a GRect' or 'archie is a GArc'. But we never say 'GRect is a fritz'.
- We can have several objects, e.g. there is a fritz and a lisa object (both are GRects), but there is only one GRect class.

To make this clear in the code, the names of objects always start with lowercase letters, while class names always start with uppercase letters.

SEP: Objects start with lowercase letters, classes with uppercase letters.

.

Review

What have we learned in this chapter? We have

- created our first graphics program,
- got to know the classes GRect, GOval, GLine, GImage, GLabel, GArc and GPolygon,
- seen how we can change the color of objects,
- seen that messages, functions, methods and commands are all the same and
- we learned to use more than one object.

But the most important thing in this chapter was that we took our first steps in object orientation.

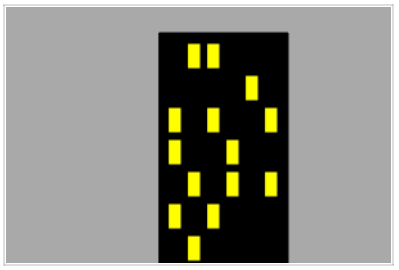
.

Projects

With our new skills, we will now realize a few interesting graphics projects. You can find many more examples in Eric Roberts's book [3].

.

Skyscraper



Try it

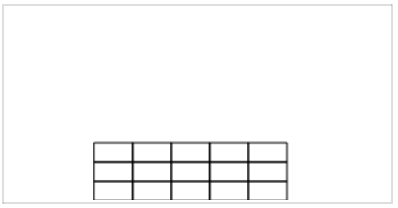
Karel loves Chicago, and Chicago is full of skyscrapers. So we want to draw a high-rise at night for Karel. Basically, it is just a few GRects. With the following line

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

we can set the background color of the canvas to dark grey. If you want you can introduce a function called drawWindows(), which is responsible for drawing the windows.

.

Wall



Try it

Next we want to build a wall consisting of 15 bricks (GRect). At the moment this is still a lot of code to write. Later we will see how to do this a lot easier.

.

.

.

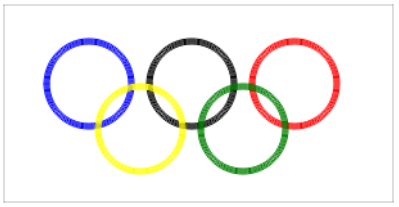
Archery

Try it

Karel likes to practice archery from time to time (remember RobinHoodKarel). So he needs a target for that. This consists of an inner red ring with 20 pixels diameter, a middle white ring with 40 pixels diameter and an outer red ring with 60 pixels diameter. If you like, you can use functions for your code to become more readable (our first SEP), namely drawInnerRing(), drawMiddleRing() and drawOuterRing().

.

OlympicGames



Try it

Archery has been an Olympic discipline since 1972, and that is why Karel is planning to qualify for the next Olympic Games. He still has to practice a lot, but in the meantime we'll draw the Olympic rings for him.

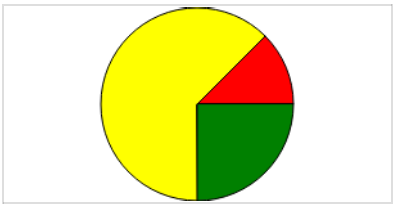
There are several approaches to solve the problem.

1. The first is to paint only the borders of the GOval, so we set setFilled(false). Then the rings are very thin and don't look quite right.
2. Paint two GOvals, one slightly larger with the desired color and the other slightly smaller with white color. However, at the intersections this looks a little funny.
3. Finally, we can paint several rings (e.g. five), each with a thickness of one, on top of each other. This looks pretty decent.

However, this is still not perfect, when comparing it to the original on Wikipedia [7]. Graphics is hard.

.

PieChart



Try it

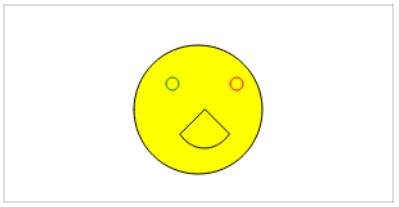
A nice application for GArcs are pie charts. In this small project we want to create a simple pie chart, consisting of three GArcs.

.

.

.

Smiley



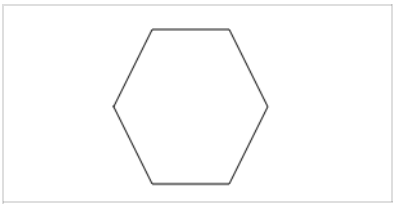
Try it

In later projects we will need a smiley face. A smiley consists of a GOval for the face, two small GOvals for the left and the right eye and a GArc for the mouth. If you draw more complicated graphics, it always makes sense to draw the design first on a piece of paper. This makes it easy to read of the coordinates.

Also with the smiley program we can use functions to make our code more readable. For example, the functions could be drawFace(), drawLeftEye(), drawRightEye() and drawMouth().

.

Hexagon



To familiarize ourselves a bit with the GPolygon class, we want to draw a hexagon. Hexagons are cool because they always remind Karel of bees, and they make Karel's favorite food: honey.

.

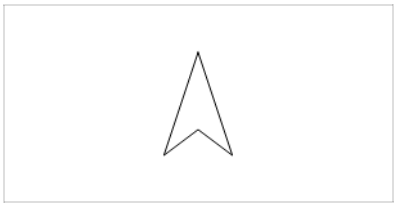
.

Try it

.

SpaceShip

In chapter seven we want to program the classic game Asteroids. For that we need a spaceship. The GPolygon is suitable for both the spaceship and the asteroids. So in this exercise we want to construct a spaceship from a polygon.



Try it

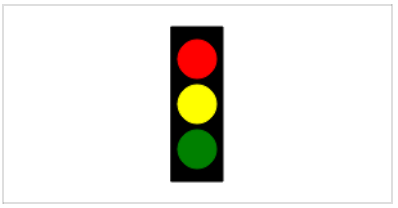
.

.

.

TrafficLight

A traffic light consists of a black rectangle and three colored circles. This can be easily constructed from one GRect and three GOvals. We'll animate the traffic light later.



Try it

.

.

.

Heart

Karel would like to draw a heart for his great love. For this we need two red GOvals and one red GPolygon. Of course they should be filled, otherwise you can see how it was constructed.

Try it

.

.

.

CarSymbol

Karel loves cars, and of course Mercedes is his favorite car. In this project we want to create a logo for our favorite car (not necessarily Mercedes) using graphics objects. The Mercedes example consists of GOvals and GArcs.

Try it

.

.

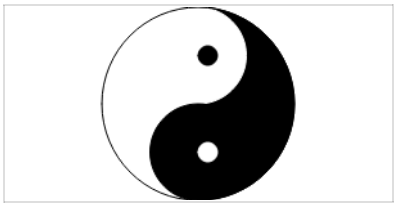
.

YinYang

Karel is fascinated by philosophy. A few days ago he read about Yin and Yang in Wikipedia [6]:

> "The highest Yin is cold, the highest Yang is hot.
> Cold springs from the sky, heat flows from the earth.
> If they penetrate each other and achieve harmony in the process,
> all things come out of it."

We can draw the YinYang symbol using GOvals and GArcs.

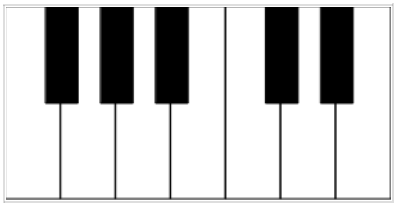


Try it

.

Piano

Of course Karel loves music. And in his spare time, he's always jingles on the piano. Therefore we draw a keyboard for him, consisting of eight white and five black keys (GRects). By chapter seven our piano will also make music...



Try it

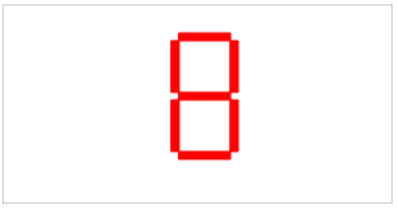
.

.

.

SevenSegmentDisplay

The seven-segment display is mainly used in digital devices to display numbers. The digits are composed of seven individual "lines", also called segments [8]. We can create a seven-segment display from seven red rectangles. Later we will also breathe life into the seven-segment display.



Try it

.

.

QUESTIONS

1. Which of the graphics classes would you use to draw a star?

2. In the following code, is 'fritz' an object or a class?
```
let fritz = new GRect(50,50);
```

3. What is the difference between setColor() and setFillColor()?

4. What are messages, what are they used for?

5. Name five graphics classes you have learned this semester.

6. What is in the graphics.js file?

7. Name three classes that can be found in the 'acm.graphics' package.

8. Write code that fills the following rectangle with green color:
```
let fritz = new GRect(50,50);
```

9. What is a polygon? Give an example for its application.

10. Draw the Audi logo using GOvals.

.

REFERENCES

Details about the ACM graphics library can be found on the pages of the ACM Java Task Force [1]. There you will also find a nice tutorial [2]. The book by Eric Roberts (who is also behind the Java Task Force) is a classic and a true treasure chest, full of examples and deep insights. Many of the examples in this chapter are inspired by his book and by the 'Programming Methodologies' lecture [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

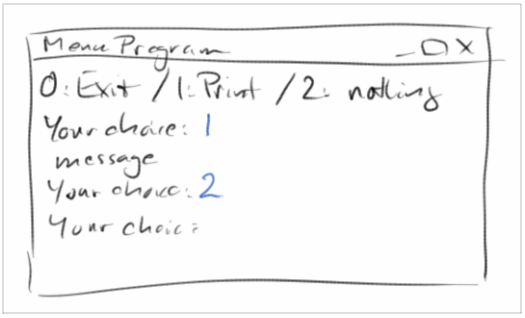
[6] Seite „Yin und Yang“. In: Wikipedia, Die freie Enzyklop�die. URL: https://de.wikipedia.org/w/index.php?title=Yin_und_Yang&oldid=149503099

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

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

.

.



CONSOLE

Console programs are actually even easier to write than graphics programs. Console programs are text based and they are about interacting with our users. Many concepts we have learned and used in the last two chapters will become clearer in this chapter and they will be set in a new context and explained in more detail.

.

Console Program

Let's have a look at a simple 'HelloWorld' console program:

```
async function setup() {
  createConsole();
  println('Hello World!');
}
```

```
Hello World!
```

Try it

As usual, we are only interested in the code inside the setup() function. It basically tells the computer to display the text in quotation marks, i.e. "Hello World", in a console window. 'println' is the short form for 'print line' and means exactly that, so write a line in the console window. Whenever we want the user to enter some information, we need to add the *async* keyword before the *setup()* function. Console programs will not work without it.

.

readInt()

Console programs would be quite boring if all we could do is 'println()'. The counterpart to println is *readInt()*. It allows the user of our program to enter a number:

```
let n1 = await readInt('Enter number one: ');
```

The text in quotation marks is not really necessary, but it gives the users an indication of what they should do. Here we see the *await* statement: it means await input from the user. If we forget it, the program wont wait for the user. The await and the async always go together.

.

Exercise: AddTwoIntegers

Let's take a look at an example. We want to add two numbers and display the result on the console.

```
async function setup() {
  createConsole();
  println('This program adds two numbers.');
  let n1 = await readInt('Enter number one: ');
  let n2 = await readInt('Enter number two: ');
```

```
This program adds two numbers.
Enter number one: 4
Enter number two: 6
The sum is: 10
```

Try it

```
  let sum = n1 + n2;
  println( 'The sum is: ' + sum );
}
```

The first line simply tells the user what the program does. Then the user is asked to enter the first number. The program waits until the user enters a number. Then it prompts the user to enter the second number. After the user has done this, we add the two numbers n1 and n2 and store the result in the variable sum. The total is then displayed in the last line.

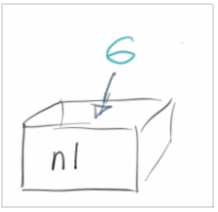
Question: What would a program look like that subtracts two numbers?

.

Variables

The question we ask ourselves immediately, what is a variable? Variables are a bit like boxes into which you can put things. What kind of things can you put in the box? For example, numbers or GRects.

Variables also have names, e.g., 'n1' or 'fritz'. We write the name on the outside of the box. For instance, if we say




```
let n1;
```

that means there's a box called 'n1'. Next we say

```
n1 = 6;
```

This is like putting the number 6 in the box. This is called assignment, so the box 'n1' is assigned the number '6'. If you want, you can change the number in the box. Then you simply make a new assignment and say

```
n1 = 5;
```

i.e., we replace the old number '6' with the new number '5'. However, the name 'n1' has not changed, it is still written on the outside of the box.

We can not only put numbers in our boxes, but also other things. E.g. with


```
let fritz;
```

we say that there is a box called 'fritz'. We can put GRects into this box, so with

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

we put a new GRect 50 pixels wide and 50 pixels high in the box.

.

Declaration and Assignment

When we say

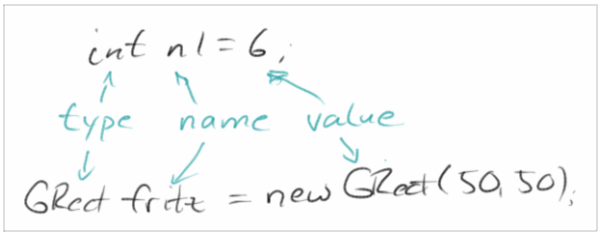
```
let n1;
```

we *declare* a variable. The variable is called 'n1' and has no value, it is *undefined*. When we say

```
n1 = 6;
```

we make an *assignment*, i.e. we assign the value '6' to the variable 'n1'. So the '=' does not mean equality, but assignment. This is very important.

As far as the names of variables are concerned, they can be almost arbitrary, they can consist of letters, numbers and the underscore. However, some words are not allowed, such as 'if' and 'for', because they are already used by JavaScript. Names should not start with numbers and special characters should generally be avoided. And you always have to worry about upper and lower case!



A variable always has a name, a type and a value.

SEP: The names of variables should always be descriptive, e.g. 'blueRect'.

.

Types

What kind of types are there? In the last chapter we already got to know some: GRect, GOval, GLine etc. are all types, more precisely data types. But there are other types too. The ones that will occupy us in this chapter are the simple data types. There are about six in JavaScript, but we are only interested in the following three:

- **number:** can be integers or floating point number.
- **boolean:** used for logical values, can only have the two values true or false.
- **string:** a string can contain letters, digits, but also special characters such as '.' and '$', etc.

To see how they are used consider the following examples:

```
let x = 42;
let y = 42.0;
let b = true;
let z = "42";
```

The first two, x and y, are numbers. The third one is a boolean. Booleans should have the values *true* or *false*. Notice there are no quotation marks around the true or false. The last defines a string, you can tell by the quotation marks. They can be single or double quotation marks, but they should match. We will deal with the boolean data type below, the string data type we will cover in the next chapter.

.

Expression

The word *expression* as we use it, is used in the sense of 'mathematical expression'. We have already seen an example, namely

```
let sum = n1 + n2;
```

On the left side of the assignment there is a variable, 'sum', and on the right side there is a mathematical expression, namely the sum of the numbers 'n1' and 'n2'. More precisely, the sum of the numbers inside the boxes 'n1' and 'n2'.

So what this line means is: Get the numbers in the boxes 'n1' and 'n2', add them together and save the result in the box 'sum'. You can illustrate this very nicely with the 'Jeliot' program [1].

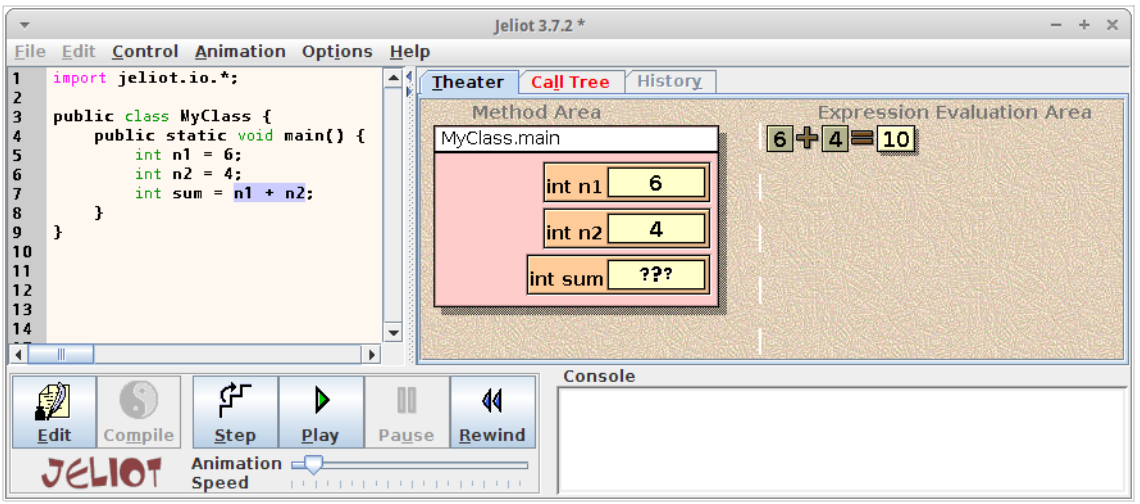
.

Exercise: Jeliot

Jeliot is a very nice program that helps us to visualize what happens, when we execute the three lines:

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

We see our code in the code window on the left. We can go through this code step by step. And on the right side we see the Method Area where we also see our boxes 'n1', 'n2' and 'sum', as well as the Expression Evaluation Area. We observe how the expression '6+4=' is currently being evaluated. In the next step the result ('10') is put into the box 'sum'.



.

Operators

When we use the word operator, we use it in the mathematical sense, i.e. the 'plus' operator or the 'minus' operator. In JavaScript we use the characters '+', '-', '*' and '/' to denote these operations.

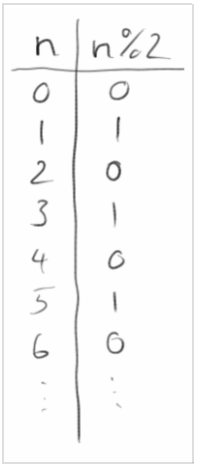
Expressions can also be a little longer:

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

In such a case, they are usually evaluated from left to right. Kind of in the same way you write them. The exception, however, are multiplication and division. You might remember from school: brackets before multiplication/division before addition/substraction.

.

Remainder



One operator we haven't talked about is the remainder operator, sometimes also called the modulo operator. The last time we probably heard about it was in the second grade, and that was before we were able to divide properly. If we calculated 5 / 2 at that time, then the result was: two *remainder* one. Or the result of 4 / 2, was two *remainder* zero. Do you still remember those times? Those were good times.

```
let remainder = 5 % 2;
```

It turns out that the remainder operator is extremely useful, so in JavaScript there is an extra sign for it, the percentage sign: '%'. Actually, we use the remainder operator every day: if we say it is 2 pm, then we have implicitly calculated 14 % 12 in our mind.

.

Exercise: Even Numbers, Odd Numbers

A useful exercise is to list the numbers from 0 to 6 and to calculate the remainder with respect to division by 2 for each of these numbers, i.e. x % 2 We see that for even numbers the remainder is always zero and for odd numbers the remainder is always one. This is very useful, because sometimes we want to know if a number is even or odd.

.

Exercise: AverageTwoIntegers

```
This program averages two numbers.
Enter number one: 2
Enter number two: 3
The average is: 2.5
```

Try it

We have to be a little careful when doing mathematical calculations, the example AverageTwoIntegers will illustrate why. Similar to AddTwoIntegers, we first read in two integers. And if someone is a bit careless, he might write the following statement to calculate the average of two numbers:

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

Interestingly, this even works, e.g. if n1=0 and n2=2. However, for other numerical values you quickly notice that something is wrong.

Of course, the error was the missing parentheses! What we should have written is the following:

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

However, an even better solution is the following:

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

In JavaScript it makes little difference, in other languages it does.

SEP: You should always test your code thoroughly.

.

Order of Precedence

There are many rules relating to the order of precedence in JavaScript. The following hierarchy lists the most important ones:

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

It says that first parentheses are evaluated, then multiplication, division and remainder, and finally addition and subtraction. But it can get even more complicated with operators like '^', '&', '|', etc. Therefore it is always recommended to make heavy use of parentheses.

SEP: To avoid trouble, use a lot of parentheses!

.

Constants

Some variables do not change, so they are actually constants. For example, the circle number 'Pi' always has the value '3.1415...'. To make a variable constant, we mark it with the keyword *const*:

```
const PI = 3.1415;
```

Constants cannot be changed once they have been initialized. This may seem not very useful at first, but later we will see that using constants leads to much better code.

SEP: Constants should always be written in capital letters, e.g. 'MAX_NUM', so that they can be immediately distinguished from normal variables.

.

.

Exercise: Area

Write a ConsoleProgram that asks the user for the radius of a circle, then calculates its area (area = PI * radius * radius) and prints the result to the console (println()). PI should be a constant in this program.

```
Calculate the area of a circle.
Enter radius: 2
The area of the circle is: 12.566
```

Try it

.

Boolean Values

So far we have dealt with the numbers. Now let's take a look at the logical type, also called *boolean*. To get a feeling for booleans, we take a look at the following inequality:

```
3 > 5
```

It is like a question: is three greater than five? The answer is no. One also says the statement '3 > 5' is wrong or false. To deal with this, the data type boolean was invented:

```
let b = 3 > 5;
println(b);
```

That is, the variable b is of the data type boolean and can take the values *true* or *false*. The whole thing should remind us a little of Karel's sensors: e.g. the beepersPresent() sensor always returned true if a beeper was there and false if none was there.

.

Conditions

Boolean expressions mostly make sense when used in a condition. In Karel's world we used them with the if statement:

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

We can do the same in a console program:

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

```
Enter a number: 6
Number is larger than 5.
```

Try it

The fact that 'x > 5' is a boolean expression is actually not really important.

SEP: We should always use the curly braces with an if statement!

.

Comparisons

What other comparisons exist besides '>'? Six in all:

```
==      equal to
!=      not equal to
```

```
>       greater than
<       less than
>=      greater than or equal to
<=      less than or equal to
```

In addition, JavaScript has two more comparison operators: the exactly equals '===' and the not exactly equals '!=='. They compare not only value but also data type. The following example shows the difference:

```
let x = 42;
let y = "42";
if (x == y) {
    println("x and y are equal");
}
if (x === y) {
    println("x and y are exactly equal");
}
```

Since x is a number and y is a string, they are equal, but not exactly equal. In general, always use the exactly equals operators.

.

Exercise: DayOfTheWeek

Let's write a program that tells us whether a day is a weekday, a Saturday or a Sunday:

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

```
Enter day of week as int (0-6): 2
Weekday
```

Try it

If the day entered is zero, it is a Sunday; otherwise, if the day is less than or equal to 5, it is a weekday; and otherwise it must be a Saturday. This form of concatenated if statements is also known as a 'cascading if'.

.

switch Statement

Karel didn't know about the switch statement, but it is quite handy. In principle, it does the same thing as the 'cascading if', but a little more elegantly:

```
let day = await 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;
}
```

If you do not think that this looks more elegantly just wait a little.

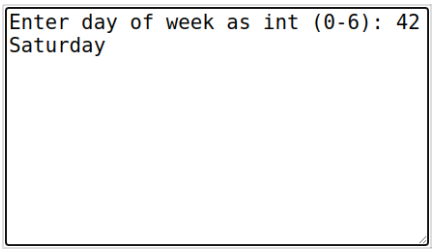
SEP: You should always have a default in your switch statement. Also, in a cascading if there should always be an else branch.

.

Exercise: DayOfTheWeek

We want to solve our DayOfTheWeek problem with the switch statement. First, we use the code as above, and test if it works. Testing means that we test many different possible inputs, even unusual ones. (What happens if we enter -1 or 42?)

Question: What happens if we omit one of the break statements?



Try it

.

Boolean Operators

Let's remember YardstickKarel: in the last step, when Karel was collecting all the beepers to put them in a big pile, it would have been great if we could have tested two conditions at the same time:

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

Here the '&&' means as much as 'and', so only if both conditions are fulfilled, Karel should do something. It would have been similarly practical if we had a way for Karel only to do something when something is not fulfilled,

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

Here the '!' means as much as 'not'. Finally, we had the example of BreadcrumbKarel where Karel was supposed to do something when there was either a wall in front of him or no beeper:

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

That '||' means 'or' in the sense of either-or, i.e., if either the front is blocked or there are no beepers present should Karel do something.

These three operators are the so-called boolean operators:

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

The order also reflects the priority rules, so '!' has a higher priority than 'and'.

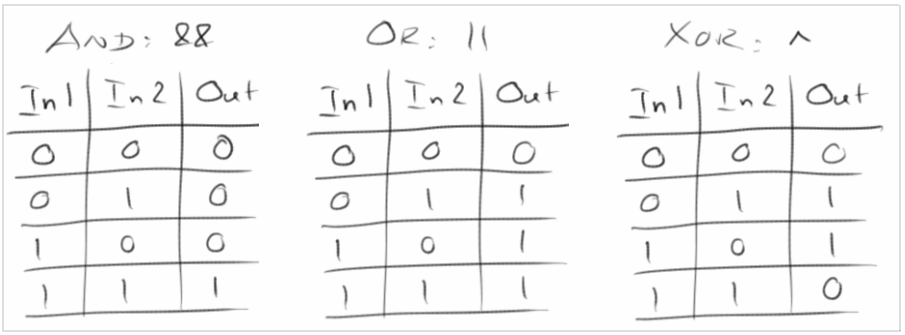
.

Truth Tables

It is common to display boolean operations in so-called truth tables. Here b1 and b2 are the two boolean operands, and out is the boolean result, e.g.

```
let out = b1 && b2;
```

for the 'and' operation. The truth tables for the three logical operations are as follows:



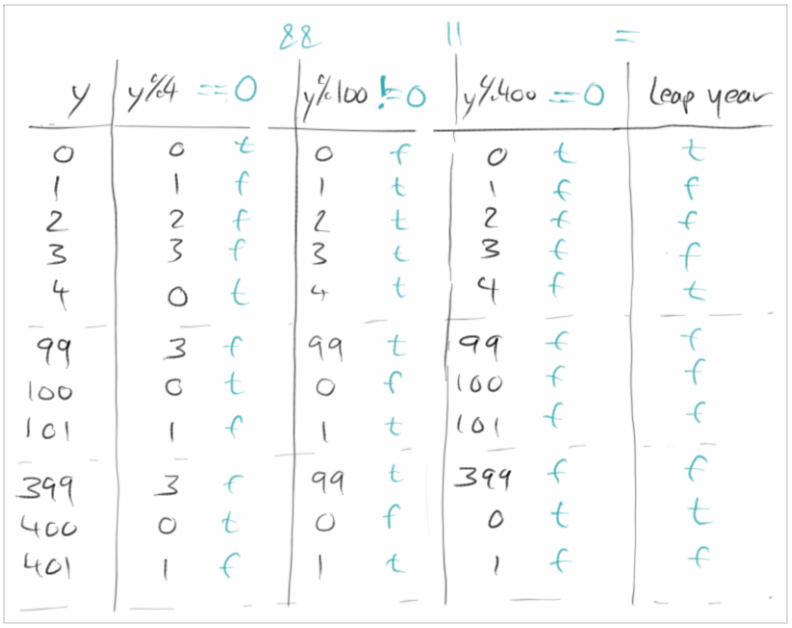
.

Exercise: LeapYear

There is a cool boolean expression that tells us whether a year is a leap year or not:

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

To understand the expression, it is best to write down the truth table for it.



.

while Loop

We have already met the while loop with Karel:

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

The while loop is executed as long as a certain condition is fulfilled. As a simple example, let's output the numbers from 0 to 9.

```
let i = 0;
while (i < 10) {
```

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Try it

```
    print(i + ", ");
    i = i + 1;
}
```

In the first line we declare a variable named i of data type int and set its value to 0. Then we test if i is less than 10. As long as this is the case, we print the current value of i in the console window. After that we increase the value of i by one.

This last line might be a bit unusual (especially mathematicians have a problem with it). But it is important to remember that '=' does not stand for equality, but for assignment. So

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

means: take the current value of i, add one to it, and then assign the result to the variable i.

.

Exercise: CounterWithWhile

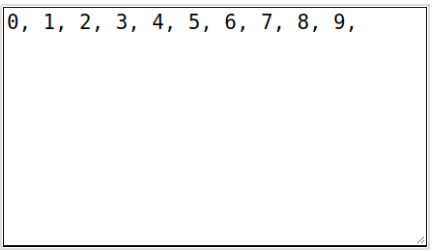
As a little exercise we will try out the counter code: simply insert the lines above into the setup() function of a console program. What happens if we only use print() instead of println()?

.

for Loop

We got to know the for loop from the very beginning with Karel. At that time, we were only interested in the number of times the loop is run. But now we are ready to understand the rest. The following for loop also outputs the numbers from 0 to 9:

```
for (let i = 0; i < 10; i++) {
    print(i + ", ");
}
```



Try it

In general, a for loop always looks like this:

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

First, the init step is executed, usually something like 'int i = 0'. Then the condition is checked, so is 'i < 10'? If yes, the statements are executed within the loop. Finally, step 'i++' is executed. This is done until the condition is no longer fulfilled.

What does this 'i++' mean? It is called the increment operator, and it means as much as the line,

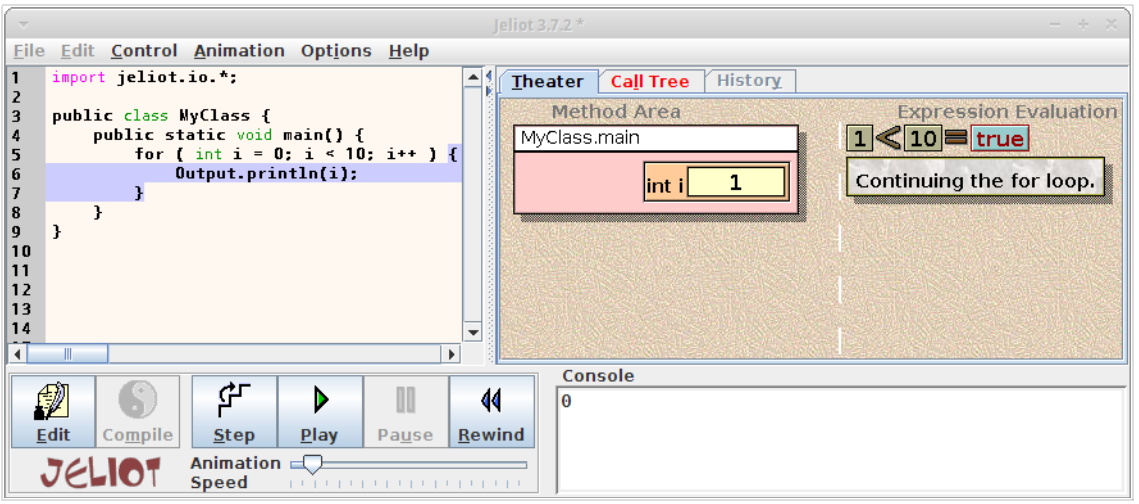
```
i = i + 1;
```

So the value of the variable i is increased by one. There is also a decrement operator that does exactly the opposite: 'i--'.

.

Exercise: For Loops with Jeliot

To understand the for loop even better, let's take a look at the for loop in Jeliot and watch step by step what happens.



.

.

for versus while

As we have seen, while loop and for loop actually do the same thing. Thus the question arises when to use which?

- **for:** if we know from the outset how often something is done, e.g.,
  `for ( let i=0; i<10; i++ ) {...}`
- **while:** we use the while loop, if we don't know exactly how often a loop is executed, e.g.,
  `while ( frontIsClear() ) {...}`

.

OBOB: FillRowKarel

There is another loop we haven't talked about yet, the 'Loop and a Half': it is the solution to our OBOB problem. Let's remember FillRowKarel:

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

The problem is that we need the 'putBeeper()' function twice. This duplication of code occurs with every OBOB and duplicate code is always a bad thing.

SEP: Avoid duplicate code.

.

Loop and a Half

The solution is actually quite simple and is called the 'Loop and a Half':

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

Let's analyze the code: First of all, we have an infinite loop, 'while (true)'. However, we can end this infinite loop with the break statement. We do this when there is a wall in front of Karel, 'if (frontIsBlocked())". Now we can see why it

is called 'Loop and a Half': only half of the loop is executed in the last step. To understand the code, we should go through it step by step on a piece of paper.

.

Exercise: CashRegister

A nice application for the 'Loop and a Half' is the cash register in a grocery store: assume, we have five items in our shopping cart and we are ready to pay. The cashier enters the prices of the individual goods, one after the other into the cash register. It adds them up and at the end outputs how much we should pay:

```
Enter price: 3
Enter price: 1
Enter price: 5
Enter price: 0
Your total is: 9
```

Try it

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

But since not all people always buy exactly five things, we need some termination criterion: in our case it is when the cashier enters 0 for the price. This abort criterion is sometimes also called the sentinel.

Question: Could we also use a negative number as abort criterion?

SEP: If possible, you should not use more than one break statement.

.

Post-Increment vs Pre-Increment

So far we have only seen the increment operator as post-increment operator, but it is also available as pre-increment. The difference is that the '++' operator is either before the variable (pre) or after (post) it. So:

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

or

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

The difference is subtle, and you only notice it when this operator is related to assignments or other operations:

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

In the second line, first (pre) the variable a is increased by one, and then the assignment is done. In the third line, the assignment is done first, and then (post) the variable x is incremented by one.

.

.

.

.

typeOf

You can ask JavaScript what the type of a given variable is with the *typeof* operator. For instance, the following code,

```
let x = 42;
println( typeof(x) );
```

would output *number* to the console. You may want to try the following examples to see how the typeof operator works.

```
let u1;
println(typeof (u1));  // undefined
let u2 = undefined;
println(typeof (u2));  // undefined
let x = 42;
println(typeof (x));   // number
let y = 42.0;
println(typeof (y));   // number
let z = "42";
println(typeof (z));   // string
let b = true;
println(typeof (b));   // boolean
let n = null;
println(typeof (n));   // object
let arr = [];
println(typeof (arr)); // object
let obj = { name: "Garfield", age: 42 };
println(typeof (obj)); // object
let f = function () { };
println(typeof (f));   // function
```

```
number
string
undefined
boolean
object
object
object
function
object
GRect
```

Try it

The last four types you have not seen, they will come soon.

.

REVIEW

We have made it! This chapter was a little more difficult than the previous two. But from now on it's all downhill. What have we learned in this chapter? We have

- got to know variables,
- reminded us of the remainder operator,
- had our first contact with constants,
- seen boolean operators and truth tables for the first time,
- found that the if and switch statements are very similar,
- learned when we should use a for and when a while loop and
- found the solution to our OBOB problem with the 'Loop and a Half'.

The most important thing in this chapter however was that we filled all the little details we left unexplained in the first two chapters.

.

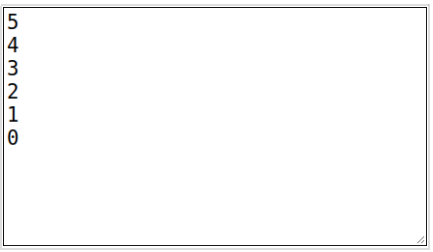
PROJECTS

The following projects may not be as interesting as those in the last chapter. But they are the basis for future chapters.

.

Countdown

Karel loves rockets and especially the countdown just before lift-off. Therefore we want to write a program that displays the numbers from 10 to 0 on the console.



Try it

.

.

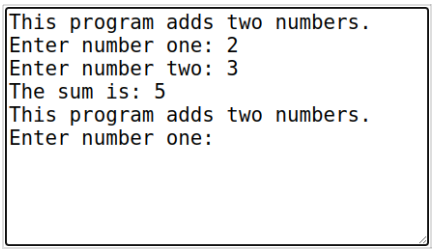
.

.

Calculator

The DoubleBeeper was a lot of work for Karel. This is much easier with console programs and variables. That's why we want to write a console program that adds two numbers.

What is a bit annoying about the program, that we have to restart it every time we want to make a new addition. What could be done to allow the program to allow multiple additions (always of two numbers)?



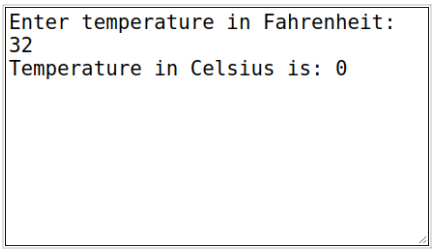
Try it

.

Temperature

Karel loves Europe, but he can't deal with those Celsius. That's why we write a program for him that converts Fahrenheit to Celsius. We ask the user to give us a temperature in Fahrenheit. Using *readInt()* we store them in the variable *f*. And with the formula

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



Try it

we can convert it to Celsius.

What happens if you enter 33 for the Fahrenheit? The result is: 0.5555555555555556. If you rather want rounded results, you can use the function *Math.round()* to do that:

```
let c = Math.round( (5.0 / 9.0) * (f - 32) );
```

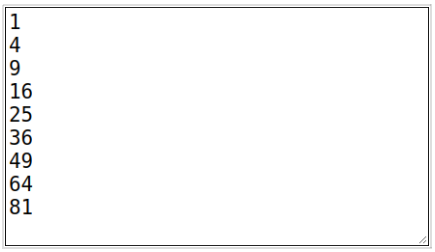
We will also need the function *Math.trunc()*: it truncates, that rounds off the result. Both can be used to convert floating point numbers to integers.

.

Squares

As an example for the use of constants, we want to write a console program that outputs the square numbers of the numbers from 0 to MAX_NUM, where MAX_NUM is a constant that is to be set to the value 10.

SEP: 'Magic Numbers' are numbers that appear somewhere in our code, and we usually have no idea where they come from and what they mean. Therefore, all numbers except 0, 1 and 2 should be declared as constants.



Try it

.

EvenOdd

We want to write a small console program that lists the numbers from 0 to 10, next to it the value of the (number % 2), and finally whether the number is even or odd. What we see is that the remainder operator can be used to determine if a number is even or odd.

```
i : i % 2 : even/odd
0 : 0 : even
1 : 1 : odd
2 : 0 : even
3 : 1 : odd
4 : 0 : even
5 : 1 : odd
6 : 0 : even
7 : 1 : odd
8 : 0 : even
```

Try it

.

.

.

Money

If we are dealing with money in our programs, should we use an int data type or a double data type? The answer is very simple when you ask yourself: can you count money? Everything you can count uses the int data type.

That means when we work with money we should always think in cents. However, if we then spend them, it would be better if we did not spend 120 cents but 1.20 euros. Here too, the remainder operator '%' is of great benefit:

```
Enter amount in cents: 120
The amount is 1,20 Euro.
```

Try it

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

To make sure that our code really works, we should try different inputs. Good test candidates are the inputs: 120, 90, 100, 102 and 002. Our program has always returned the correct output, yes?

.

BigMoney

For amounts over a thousand euros, one more comma is added for the thousands and millions, for example: 1,001,233.45 euros. So we want to write a function formatNumericString(int cent), which gets an amount of money in cents as parameter, and returns a string that is correctly formatted. When we test the program, we should use the above amount. We will find that it makes sense to write a function padWithZeros().

```
Enter amount in cents: 100123345
The amount is 1.001.233,45 Euro.
```

Try it

Also here we should try different inputs, e.g. 100123345, 100123305, and 05 might be good test candidates.

.

Time

People prefer to specify the time in hours, minutes and seconds. Computers, on the other hand, only think in seconds, seconds that have passed since midnight. That's why we want to write a program that calculates the current time from the seconds that have passed since midnight. For this we again need the remainder operator. It may also make sense to write a function padWithZeros() which ensures that "06" minutes are displayed instead of "6" minutes.

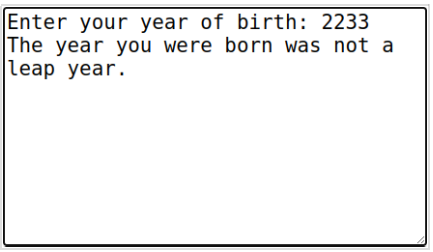
```
Enter time in seconds since
midnight: 34522
Time is: 09:35:22
```

Try it

For testing we should try at least the following inputs: 5, 61, 85, 3600, 3601.

LeapYear

We have seen above how to determine if a year is a leap year. With this knowledge, we want to write a console program that tells us if the year someone was born in was a leap year. For example, 1996, 2000, and 2004 should be leap years, but 1800 and 1900 should not be.



Try it

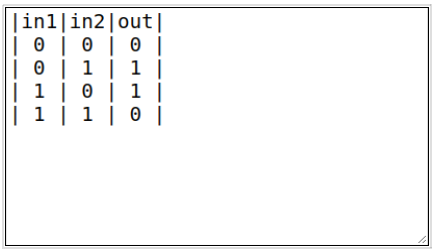
.

.

.

TruthTables*

In this project we want to deal with the boolean data type and the logical operators && (and), || (or) and ^ (exclusive or). We want to write a program that calculates and outputs the truth tables for all three operators. Possibly helpful is the following trick how to create a boolean datatype from an number datatype:



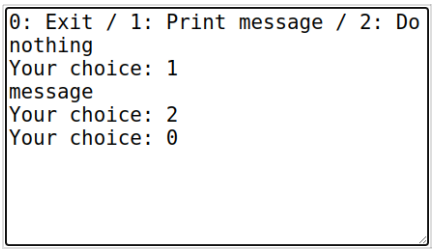
Try it

```
let in1 = 0;
let b1 = (in1 != 0);
```

.

InteractiveMenuProgram

Very often you need a kind of menu in a console program. The user can choose between different menu items by entering a number. We therefore want to write a program in which the user can choose between three possibilities: if he enters '0', the program should be terminated, if he enters '1', a message should be output, and if he enters '2', nothing should happen. The first thing you want to do is, tell the user what the options are:



Try it

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

Then comes a 'Loop and a Half', in which we ask the user for the choice:

```
let choice = await readInt("Your choice: ");
```

If the user enters the '0' we end the 'Loop and a Half':

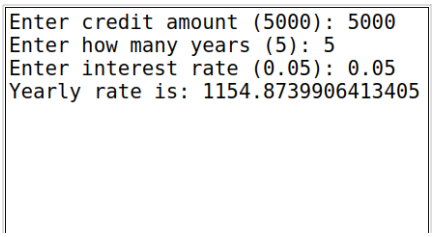
```
if (choice == 0) break;
```

After this follows a switch or cascading if for the options '1' and '2'.

.

YearlyRate

Karel wants to buy a car, so he started saving up. He wants to buy a Mini (Mercedes is to expensive), and he has seen a used one for 5000 euros. He saw a cheap loan from a bank of 5% a year. What's his annual rate if he wants the loan to be paid off in five years?



For this Karel needs a program in which he can enter the loan amount (k), the interest rate (z) and the years (n). The program should then tell him how high his annual rate (y) is. The formula for this can be found in the Wikipedia under compound interest formula [2]:

Try it

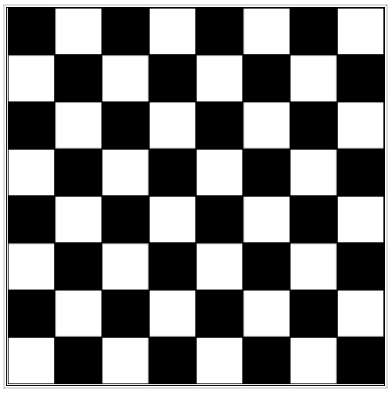
```
let q = 1.0 + z;
let qn = Math.pow(q, n);
let y = k * qn * (q - 1) / (qn - 1);
```

It is helpful to know that floating point numbers can be read with readDouble().

.

ChessBoard

We want to draw a checkerboard pattern using GRects. Of course, this must be a GraphicsProgram. It is a nice, but not quite easy application of loops and residual operator '%'.



Try it

.

.

.

.

.

.

.

QUESTIONS

1. Name three of the standard data types of JavaScript.
2. What is a magic number?
3. What is the difference between a graphics program and a console program?
4. In the lecture you have heard of many software engineering principles. Give three examples.
5. During programming there are some typical errors that happen again and again (Common Errors). Name two of those errors.
6. Write a console program that asks the user for the radius of a circle, then calculates its area (area = PI * radius * radius) and prints it to the console.
7. What is the difference between 'print("hi")' and 'println("hi")'?
8. What does the 'readInt()' function do?
9. What are the values of the variables 'x' and 'y' after the following three lines have been executed?
   ```
   a = 6;
   x = ++a;
   y = x++;
   ```
10. Describe in your own words what the following expression means:
    ```
    (x < 0) || (x > WIDTH)
    ```
11. What is the advantage of using constants?

12. Compare the two 'switch' examples below, and describe their output.

a)
```
let day = 0;
switch (day) {
case 0:
    println("Sunday");
    break;
case 6:
    println("Saturday");
    break;
}
```

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

13. Which values can a boolean variable take?

14. What data type must the variable 'b' be in the example below?
```
if ( b ) {
    println("hi");
}
```

15. Write the test which determines whether a number y is divisible by 400.

16. Recall the remainder '%' operator? Give two examples of what you can use it for.

17. People prefer to specify times in hours, minutes and seconds. Computers prefer to calculate in seconds. For example, the time 13:35:12 for the computer simply is 48912 seconds. Use this knowledge to convert the computer time 't' into a human readable form, i.e. hours, minutes and seconds. You can also use several intermediate steps.

18. When should you use a for loop and when a while loop?

19. Anything you can do with a 'for' loop can also be done with a 'while' loop and vice versa. Rewrite the following 'for' loop as a 'while' loop.
```
for (let i=0; i<5; i++) {
    println(i);
}
```

20. Describe the structure of the 'loop-and-a-half'. Which problem does it solve?

21. Loop and a Half: Rewrite the following Karel example using the Loop and a Half. What is the advantage of the Loop and a Half?
```
function run() {
  while ( frontIsClear() ) {
    putBeeper();
  }
  putBeeper();
}
```

.

.

.

REFERENCES

References for the topics in this chapter can be found in practically every book on Java, especially of course 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 the last chapter we have neglected the top-down approach at the expense of variables, but in this chapter we will see how to get back to the top-down approach with functions. And we will write our first animations and games. It promises to get interesting.

.

Functions

We have known functions for a quite a some time now. With Karel we called them commands, e.g. move(), turnLeft() and frontIsClear(). Also in graphics programs we used them, at that time we called them messages that we send to a GRect, e.g. setColor() and setFilled(). Even with console programs we had them, e.g. readInt() and println(). So nothing new.

With Karel there was the cool possibility to create new commands, like turnRight() or moveToWall(). That's what we did back then:

```
function turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
}
```

Wouldn't it have been cool if we could have had the following new command in our Archery program?

```
function drawCircle( radius, color ) {
        ...
}
```

Or for our Wall program, a function like the following would have been quite practical:

```
function drawRowOfNBricks( numberOfBricks ) {
        ...
}
```

Well, the nice thing is, this actually is possible!

SEP: Functions always do something, they convey an action, so functions should always be verbs.

.

Defining Functions

Creating a new function is as easy as teaching Karel new commands. The general syntax of a function declaration is as follows:

```
function name( parameters ) {
        ... body...
}
```

where we mean by

- **name:** the name of the function, here the same rules apply as for variables and function names should always be written in lower case,
- **parameters:** parameters are new, they did not exist with Karel, but they are very practical as we will see soon.

.

Exercise: Archery

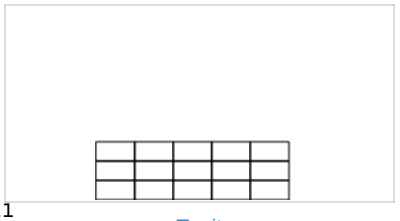
Let's take another look at our Archery program. But now we try using the drawCircle(radius, color) function. We will find that the code becomes much shorter and also more readable.

```
function drawCircle(radius, color) {
    let ring = new GOval(2 * radius, 2 * radius);
    ring.setColor(color);
    ring.setFilled(true);
    let x = 50 + 72 - radius;
    add(ring, x + 25, x - 48);
}
```

Try it

.

Exercise: Wall



Let's try the Wall program again, using the help of functions. We want to build a wall consisting of 15 bricks (GRect). However, we want to use the function

```
function drawRowOfNBricks(j) {
    let x = 70; // beginning x position of wall
    let y = 108 + j * 15; // beginning y position of wall
    for (let i = 0; i < 5; i++) {
        let brick = new GRect(30, 15);
        add(brick, x, y);
        x = x + 30;
    }
}
```

Try it

Question: Does the function above contain magic numbers? Can we change that?

.

Return Value

So far we have only seen functions that return nothing. But there are also functions that give something back. Most of the time the functions calculate something and return the result as a return value. A nice example is the following function that returns how many inches there are in a given number of feet:

```
function feetToInches(feet) {
    let inches = 12 * feet;
    return inches;
```

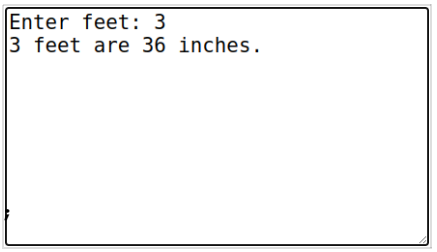
```
}
```

We can use this function in a console program.

.

Exercise: ConvertFeetToInches

We want to write a small console program that converts feet to inches. It asks the user for the number of feet using readDouble() and then calls our feetToInches() function and returns the result in the console window.




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

Try it

If we look closely, we see that we use the feetToInches() function in the same way as the readInt() or println() functions. The only difference is that we wrote it ourselves.

.

Objects as Return Value

We can not only return numbers, but every possible data type, including GOvals for instance. The following function generates a colored filled circle with a given radius r at positions x and y:

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

All we have to do now is add it to our Archery program.

.

Local Variables

The variables we have been dealing with so far are called local variables. Local with respect to a function. This means that variables are only visible inside the function in which they were declared, outside, that is, in other functions they are not visible or accessible. Let's look at an example:

```
async function setup() {
    createConsole();

    let feet = await readDouble("Enter feet: ");
    println(feet);
}

function feetToInches() {
    let inches = 42;
    println(inches);
}
```

In this example there is a variable called feet that exists inside the function setup() and a variable inches that exists in the function feetToInches(). If we try to access the variable feet in the function feetToInches(), then this is not possible. The same applies the other way round. The variables are therefore only visible locally in their respective functions.

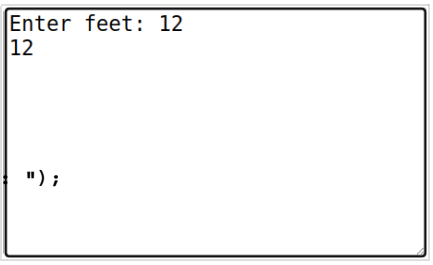
How can we pass variables from one function to another function? That's what parameters are for:

```
function feetToInches(feet) {
    let inches = 12 * feet;
    println(inches);
}
```

We pass variables from one function to another using these parameters.

However, only a copy of the variable is passed. To see this, let's have a quick look at the following program:

```
async function setup() {
    createConsole();

    let feet = await readDouble("Enter feet: ");
    feetToInches(feet);
    println(feet);
}

function feetToInches(feet) {
    feet = 42;
}
```



Try it

First we ask the user to give us a value, e.g. '12'. We pass this value to the feetToInches() function. Now inside that function, we reassign feet to a new value, say feet = 42. But this reassignment only applies to the copy, the original remains unchanged, as we see when we look at what is displayed in the console window. The original feet still has the value '12', i.e., it was not changed.

That's why it doesn't really matter whether we call the red feet also feet or give it another name:

```
async function setup() {
    createConsole();

    let feet = await readDouble("Enter feet: ");
    feetToInches(feet);
    println(feet);
}

function feetToInches(fritz) {
    fritz = 42;
}
```

This will hopefully make clear why we need return values. Because if something is calculated in the function feetToInches(), then this is only visible locally inside that function. To get it back out into the calling function, we need the return value.

As a note, a function can have several parameters, but it can only have one return value.

.

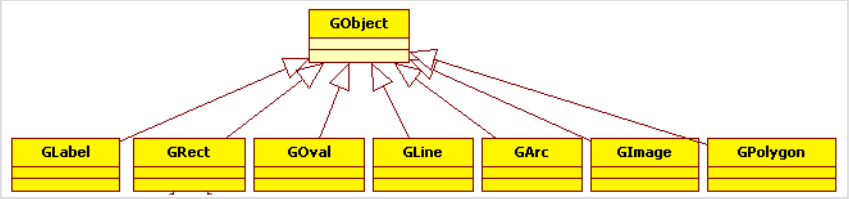
Animation

Now we have functions in our toolbox. That's great, because we can start doing really cool things with them, namely animations and games. We'll start with animations. But for that we need one more thing.

.

GObject

Let's remember the graphics classes we know already: GRect, GOval, GLine, GImage, GLabel, GArc and GPolygon. These classes are not completely independent of each other, they even have something in common: they are all GObjects.



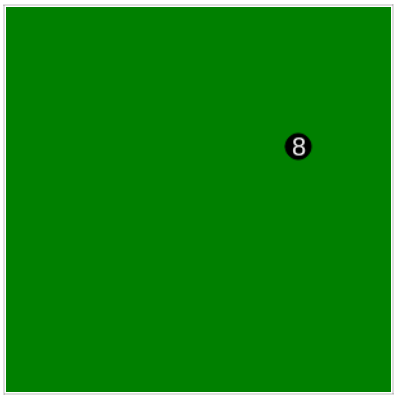
The class GObject is also called a parent class. We say that the child classes such as GRect and GOval inherit from their parent class. What do they inherit? The functions of the parent class. That's super convenient, as we'll see. But first let's take a look at those inherited functions:

- **setLocation(x, y):** move the GObject to the position x, y.
- **move(dx, dy):** move the GObject by dx and dy.
- **getX():** returns the x coordinate of the GObject.
- **getY():** returns the y coordinate of the GObject.
- **getWidth():** returns the width of the GObject.
- **getHeight():** returns the height of the GObject.
- **setColor(col):** changes the color of the GObject.

All the graphic classes we've seen so far have these functions.

.

Animations using the Game Loop



Try it

For our first animation we choose billiard: We want a black ball to move across a green table and bounce back from the sides.

Every animation has a game loop. The game loop is our *draw()* function, which we have been ignoring up to this point:

```
let ball;

function setup() {
    createCanvas(300, 300);
    frameRate(25);
    setBackground(Color.GREEN);

    ball = new GOval(20, 20);
    ball.setFilled(true);
    add(ball, 150, 150);
}

function draw() {
    ball.move(4, -3);
    update();
}
```

Everything that has to do with the setup and initialization goes in the setup() function. Then the game loop starts: if we set the frame rate to 25, then the draw() function is called 25 times per second, or every 40 ms.

Let us go through the functions in detail. In the setup() function we set the size of the window, then we set the framerate to 25 and the background color to green. Finally, we create a ball:

```
function setup() {
    createCanvas(300, 300);
    frameRate(25);
    setBackground(Color.GREEN);

    ball = new GOval(20, 20);
    ball.setFilled(true);
```

```
    add(ball, 150, 150);
}
```

What should stand out here is that it is "ball =" and not "let ball =". More on that in a moment. The next function is the draw() function:

```
function draw() {
    ball.move(4, -3);
    update();
}
```

We simply tell the ball to move. And the update() we need for the drawing to happen.

So what is the problem with the ball? Well, remember local variables: a variable declared inside a function is only valid in that function. So if we declare ball inside setup(), then it is only available inside setup(), but not draw(). If we declare it in draw(), then we create 25 balls per second, because draw() is called 25 time per second. But we do not need 25 or more balls, we need only one. We need one that is shared by both functions, or that can be accessed from within both functions. The solution is simple: just declare the ball outside the functions. This is what we call a *global* variable.

.

Exercise: Billiards

Once we know about global variables, we might as well us them. We need a ball and its velocities, hence we declare them as global variables:

```
// global variables
let ball;
let vx = 4;
let vy = -3;
```

The setup() function stays the same as above. But we want to change the draw() function a little. First, we want to return to our top-down approach:

```
function draw() {
    moveBall();
    checkForCollisionsWithWall();
    update();
}
```

Meaning we introduce two functions, called moveBall() and checkForCollisionsWithWall(). The first one is easy:

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

The second one is a little more complicated:

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

We get the current x- and y-position of the ball and test if it is inside the playing field. If not, we change the sign of the speed, i.e., the ball turns around. If you want, you could reduce the speed a little with every collision, but we don't.

Let's talk a little more about ball, vx and vy. So far we only know about local variables. The problem with local variables is that they are only valid within one function. In our billiard example, however, we need the ball in three functions: the setup(), the moveBall(), and the checkForCollisionsWithWall() functions. Obviously we cannot use a local variable for the ball (and also vx and vy). Instead, we use a global variable. Global variables are declared at the beginning of a class, before the setup() function, and most importantly, outside the setup() function (or any other function). The advantage of global variables is that they are accessible in any function. However, global variables are a little dangerous, as we will see later, hence the following SEP.

SEP: When possible try to use local variables.

.

Events

So animations are not really that difficult. Let's move on to games: our games should be controllable with the mouse. To do this, we have to tell our program that we are interested in mouse events such as whether the mouse button was pressed or the mouse was moved. In code all we have to do add the mousePressed() function to our program:

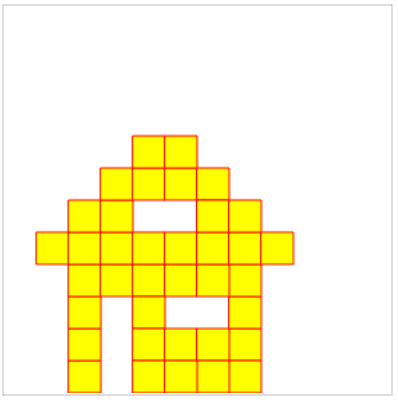
```
function mousePressed() {
    let x = mouseX;
    let y = mouseY;
    ...
}
```

The variables mouseX and mouseY contain the x- and y-position of the mouse. Similarily, if we want to find out if the mouse has moved, we implement the mouseMoved() function. Of course we can also use both. On mobile devices, like Android or iPad, the corresponding functions are called touchPressed() or touchMoved().

.

Exercise: Builder

Builder is inspired by Lego: we have small blocks and we can place them anywhere on the screen by clicking with the mouse on the position where the block should go. The program code for this is totally simple. The setup() and draw() functions do not really do anything:



Try it

```
function setup() {
    createCanvas(WIDTH, HEIGHT);
    frameRate(5);
}

function draw() {
    update();
}
```

But in the mousePressed() function all the action happens:

```
function mousePressed() {
    let x = mouseX;
    let y = mouseY;
    x = Math.trunc(x / BLOCK_SIZE) * BLOCK_SIZE;
    y = Math.trunc(y / BLOCK_SIZE) * BLOCK_SIZE;

    let block = new GRect(BLOCK_SIZE, BLOCK_SIZE);
    block.setColor(Color.RED);
```

```
    block.setFilled(true);
    block.setFillColor(Color.YELLOW);
    add(block, x, y);
}
```

To get the x- and y-position of the mouse we use the mouseX and mouseY variables. Once we have those, we create a new GRect and place it at the current mouse position.

With a little trick, we can "quantize" the position of the blocks:

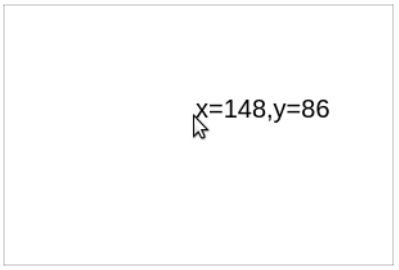
```
x = Math.trunc(x / BLOCK_SIZE) * BLOCK_SIZE;
```

This is a trick you will see again and again.

.

Exercise: MouseTracker

Next we want to track the mouse movement. We start again with the setup() function:



Try it

```
// global variables
let lbl;

function setup() {
    createCanvas(300, 200);
    frameRate(25);

    lbl = new GLabel("");
    lbl.setFont('Arial');
    lbl.setFontSize(20);
    add(lbl, 0, 0);
}
```

To display the position of the mouse we want to use a GLabel called lbl. It is an global variable and must be initialized and added to our canvas. After that we tell our program again that we want to listen to mouse events. In this example we want to know if the mouse has moved, so we overwrite the mouseMoved() function:

```
function mouseMoved() {
    let x = mouseX;
    let y = mouseY;
    lbl.setLabel("x=" + x + ",y=" + y);
    lbl.setLocation(x, y);
}
```

We get the x and y position of the mouse, change the text of the label with the setLabel() function, and move the label to the mouse position with the setLocation() function. And that's it.

.

RandomGenerator

For many games we need random numbers. To get random numbers we use the class RandomGenerator. This class does not only generate random numbers, but it can also generate random colors. The code

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

shows how to generate random numbers and random colors. Two other useful functions are *nextInt(low,high)* and *nextBoolean()*.

.

Canvas

At this point it makes sense to talk a little more about our canvas. We have been using it all the time, when we were adding our GObjects to our programs, like

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

The question we should have asked ourselves back then was: where to do we add those GRects and GOvals? The answer is of course: to the canvas. The canvas is our felt board from kindergarten. If we can add something, the next question is, can we also remove something? Or are there other things we can do with the canvas? Well, the following lists the things we can do with and to the canvas:

- **add(object):** add a GObject to the canvas.
- **add(object, x, y):** add a GObject to the canvas at position x, y.
- **addAtEnd(object, x, y):** add a GObject to the canvas at position x, y.
- **remove(object):** remove the GObject from the canvas.
- **removeAll():** remove all GObjects from the canvas.
- **sendToFront():** bring the GObject to the front (z-order).
- **sendToBack():** send the GObject to the back (z-order).
- **getElementAt(x, y):** return the first GObject at the position x, y, if there is one.
- **getElementsAt(x, y):** return all GObjects at the position x, y, if there is any.
- **setBackground(c):** change the background color of the canvas.

.

Review

It may seem that we didn't do all that much in this chapter. But that's not true at all: as we will see in the projects, we can already program really cool things. We now know what

- functions,
- parameters,
- return values,
- and local variables

are. In addition, we learned more about

- GObjects,
- the canvas,
- and the RandomGenerator.

The most important thing in this chapter however was that we can do animations using the game loop and listen to mouse events.

.

Projects

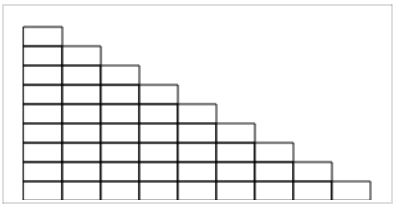
The projects in this chapter are real fun. Let's get started.

.

.

.

Stairs



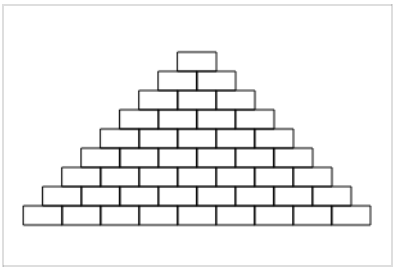
Try it

Our first project is a small staircase. The problem is very similar to the Wall problem. Therefore, it makes sense to adopt the top-down approach here as well - write a function called drawRowOfNBricks(n). We should also be careful not to use magic numbers, only constants.

.

.

Pyramid



Try it

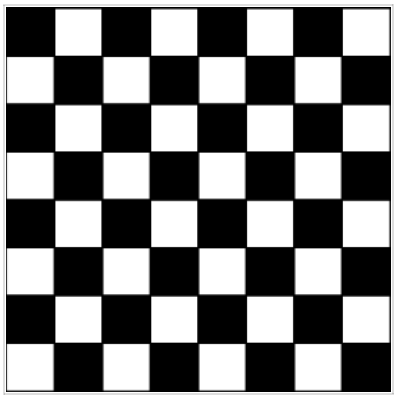
The pyramid is almost identical to the stairs. The only difference is that the steps are always offset by half a stone. The pyramid should have nine stones in the bottom row. For this, we need to change the code from the last example only slightly.

.

.

.

ChessBoard



Try it

Let's get back to our chessboard. But this time we want to use the top-down approach. There are several approaches, but one would be to declare a function called drawOneRow(). Here you have to consider exactly which parameters you pass to the function. You could also have two functions, one for even lines and one for odd lines.

.

.

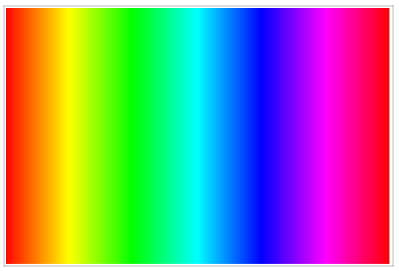
.

.

.

.

RGBColor



Try it

We have already drawn the rainbow, but still rather "manually". Now we want to draw the HSV color palette [5]. In JavaScript you can create any color using

```
let col = color(r, g, b);
```

where the variables r, g, and b each represent the red, green and blue components. These must have values between 0 and 255. So for instance, red color can be made like this

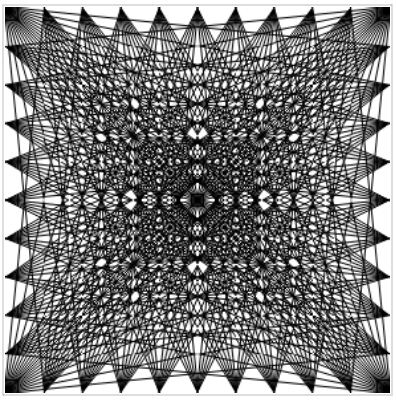
```
let colRed = color(255, 0, 0);
```

If we look closely at the colors in the HSV color palette, we notice that it starts with red, then yellow, green, cyan, blue, magenta, returning to red. Hence, there are a total of six color transitions. The first transition from red to yellow could be reached with the following lines:

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

.

Moire



Try it

The Moiré effect [6] is usually rather undesirable, but it can also be quite pretty. First we divide the length and width into equal parts, e.g. eleven parts. Then we draw a line from each point above to each point below and the same from left to right. To get a feeling for how this works, take a piece of paper and try to draw it by hand. It comes down to two nested for loops.

.

.

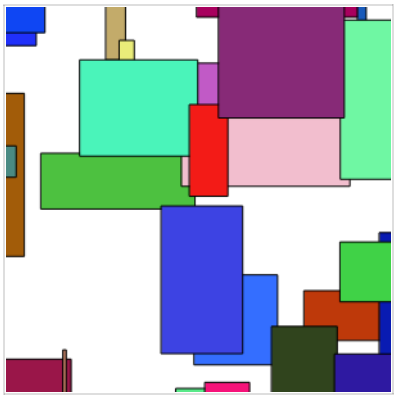
.

.

.

.

RandomSquares



Try it

Let's continue our artistic activities. We want to draw rectangles of different colors of random size and position. Of course we use the random generator. First we set a random width and height for a GRect. Width and height should perhaps not be too large or too small. Then we give the rectangle a random color with

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

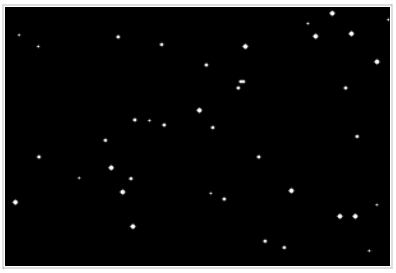
Finally, we place the GRect at a random x and y position. We place this code in the draw() function and maybe change the frame rate to 10 or less.

.

.

.

TwinkleTwinkle



Try it

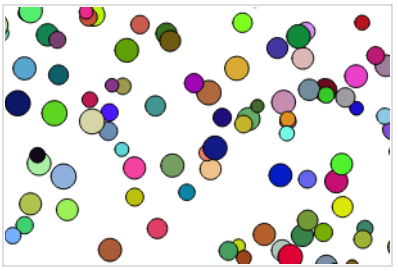
TwinkleTwinkle is about generating a random night sky. The stars are GOvals with a random size between 1 and 4 pixels. They're distributed randomly on the canvas. It makes sense to first set the background to black using

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

The whole thing then comes into the game loop and we wait perhaps 500 ms until we draw the next star.

.

Confetti



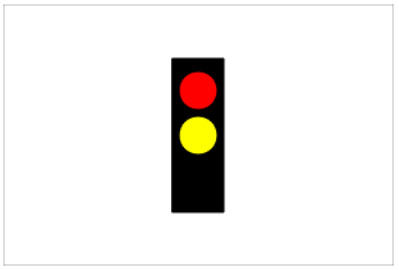
Try it

We all love confetti. They are very easy to make, either with a hole punch or with GOvals. The confetti can all be the same size (e.g. 20 pixels), but do not have to be. They have different colors, again a case for the random generator. And of course the position of the confetti should be random, and the whole thing again runs in the game loop, that is the draw() function.

.

.

TrafficLight



Try it

We also drew the traffic light two chapters ago. Now let's animate the traffic light. The traffic light starts with red, then turns to red-yellow, followed by green. Finally it goes via yellow back to red. The transition should take a second. For example, you could have global variables for the lights, and then use

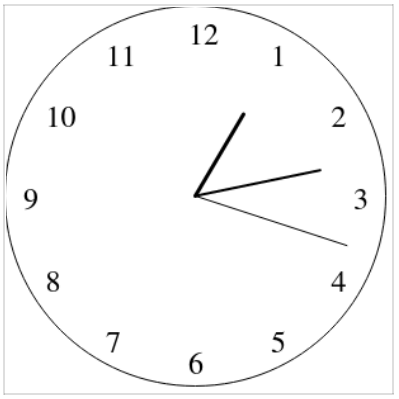
```
if (currentLight == 0) {
    redLight.setFillColor(Color.RED);
    yellowLight.setFillColor(Color.BLACK);
    greenLight.setFillColor(Color.BLACK);
} ...
```

to turn the lights on and off. Next you have to think about how to switch between the different states. This can be done very cleverly with the remainder operator %:

```
currentLight++;
currentLight = currentLight % 3;
```

.

AnalogClock



Try it

For our clock, we use a GOval for the face and GLabels for the digits. We do this in the setup() function. We could set the GLabels by hand, or calculate their position by means of sine and cosine. Both take about the same time, but the second requires a little more brains.

However, when it comes to drawing the hands, we can no longer ignore sine and cosine [7].

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

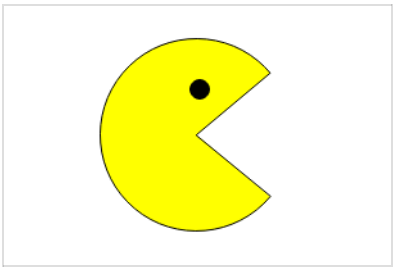
here the hand for the seconds turns out to be a Gline,

```
secondsHand = new GLine(SIZE / 2, SIZE / 2, SIZE / 2, SIZE / 2);
```

which has been declared as an global variable. How to get hours, minutes and seconds we have already seen in the project "Time" in the last chapter. Since the clock should be animated, we need to place the code in the draw() function, with a frame rate one per second, or maybe only half a second.

.

AnimatedPacman



Try it

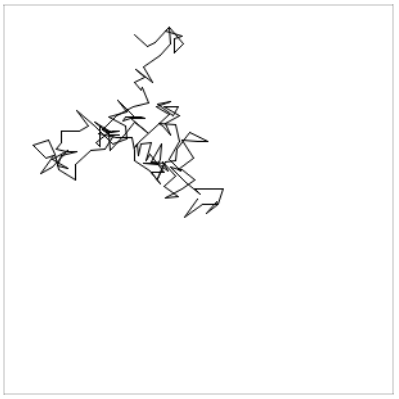
We painted our first PacMan two chapters ago. But it was quite static. We want to animate PacMan now. It is useful to know that GArcs have the following two commands:

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

If we now let the angle variable vary between 0 and 40, and do this every 50 ms, then it appears as if PacMan is animated.

.

DrunkenWalk



Try it

Once in a while Karel meets with a couple of friends in his favorite bar. On his way home, he no longer walks in straight lines. In this example, Karel starts in the middle. Once per second he takes one step, a random distance in a random direction. We connect the steps with GLines. The next morning, we show Karel his serpentines to let him know that next time he better take a taxi.

.

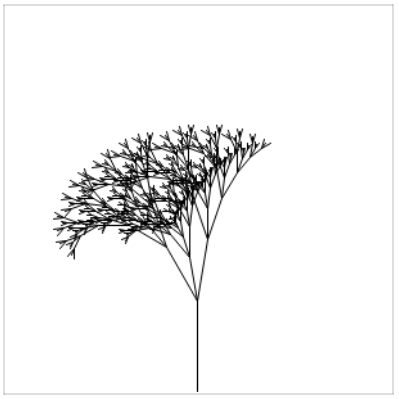
.

.

.

.

Tree*



Try it

Drawing trees turns out to be relatively difficult. A popular technique to solve this problem is recursion. Since we haven't heard anything about recursion yet, we try to draw a tree without recursion. After this experience, we may be more motivated to learn the secrets of recursion in the next semester.

.

.

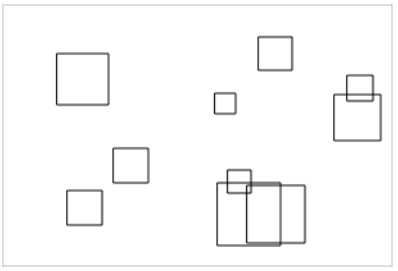
.

.

.

.

AsteroidCreator



Try it

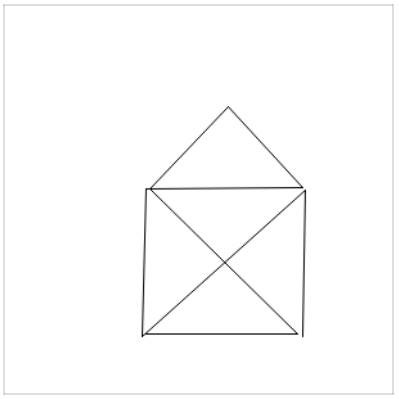
The arcade game 'AsteroidCreator®' was an absolute hit in the late 80s. The objective of the game is to draw an asteroid at the location where the user clicks with the mouse. Asteroids are simply GRects of different, random sizes with a black border. For this we have to implement the mousePressed() function, where we simply draw a rectangle at the position where the user clicked with the mouse.

.

.

.

ConnectTheClicks



Try it

Similar to the previous game, 'ConnectTheClicks®' was very popular in the late 1970s. This is a game in which the user clicks with the mouse on a point, which is then connected to the previous point. What makes the game a little more complicated is that we have to remember where the user clicked before. To do this, we simply use two global variables:

```
let x0 = -1;
let y0 = -1;
```

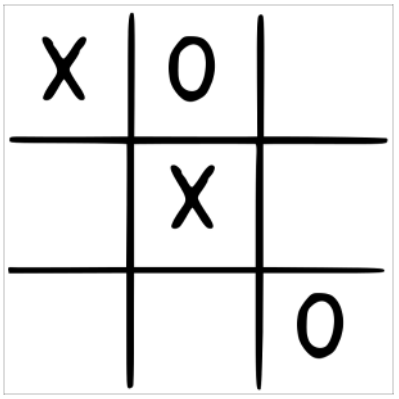
If we initialize these variables with the value "-1", we can use this to determine whether this is the first click. Because then we shouldn't draw a line. Otherwise, we simply draw a line (GLine) with each click from the old position to the new mouse position.

.

.

.

TicTacToe



Try it

Everyone knows TicTacToe from kindergarten or school, if you don't, you can read the following about it in the Wikipedia [9]:

"TicTacToe is a pencil-and-paper game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The X player usually goes first. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row wins the game."

In the setup() we draw the background. The easiest way is as an image, but we can also draw lines. Also, we have to add the mousePressed() function. There we simply have to draw an "X" or an "O" alternately, depending on who's turn it is. How do you know who's turn it is? This can be done using an global variable, for example:

```
let currentPlayer = 1;
```

This variable can have two values, '1' for player one and '2' for player two. Switching between the two players is then very easy:

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

A little thing that is very practical: you can of course simply paint the "X" and "O" where the user clicked. That looks a bit squeaky. A little trick uses Math.trunc() for positioning the "X" and "O":

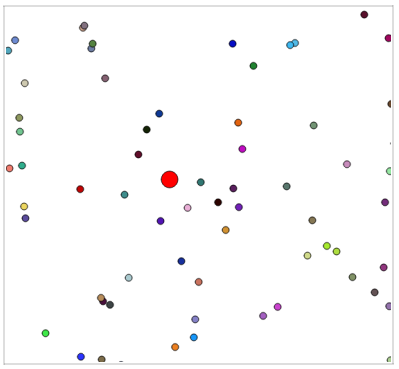
```
function mousePressed() {
    let x = mouseX;
    let y = mouseY;
    let i = Math.trunc(x / CELL_WIDTH);
    let j = Math.trunc(y / CELL_WIDTH);
    ...
}
```

.

CHALLENGES

.

Agrar



Try it

Agrar is inspired by the online game Agar.io, which according to Wikipedia [10] is about

"... navigating a cell that grows by eating pellets and other cells. However, it must not be eaten by larger cells itself."

Our version of the game is a bit simpler, there is only one cell, and it can only eat pellets. But that is already something.

First, we need an global variable for the cell:

```
let cell;
```

We initialize it in the setup() function, . Next, we need the game loop:

```
function draw() {
    moveCell();
    createRandomFood();
    checkForCollision();
    update();
}
```

Here we create a pellet at a random position, i.e., a GOval with random color. Next we check if there was a collision between the cell and a pellet. How do we know if there's been a collision? The getElementAt() function is used for this:

```
let collisonObject = getElementAt(x, y);
if ((collisonObject != null) && (collisonObject != cell)) {
    let w = cell.getWidth();
    cell.setSize(w + 1, h + 1);
    removeObj(collisonObject);
}
```

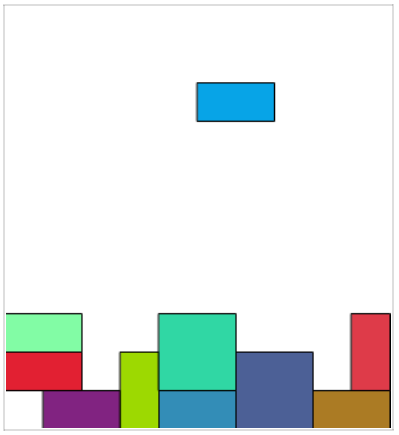
This function checks if there is something at the x,y position. If there is nothing there, the function returns the value "null". Otherwise, it returns the object that is at this position. It could be a pellet or it could be the cell itself. Therefore, we must check that it is not "null" and that it is not the cell itself. Since there is nothing else, we now know that the collisonObject must be a pellet. We "eat" the pellet, which means that the cell gets fatter and the pellet is removed.

Of course we still have to implement the mouseMoved() function. This is quite easy, we simply move the cell to the position of the mouse:

```
function mouseMoved() {
    xMouse = mouseX;
    yMouse = mouseY;
        cell.setLocation(x, y);
}
```

.

Tetris



Try it

Tetris is a classic, everybody knows it. Originally it was programmed by Russian programmer Alexei Paschitnow [11]. Tetris has different stone forms that resemble Latin letters (I, O, T, Z and L). Players must rotate the individual pieces that fall from the top of the board in 90 degree increments and move them so that they form horizontal, preferably gapless rows at the bottom edge. As usual, we limit ourselves to a somewhat simpler version, in which there are only four stone forms: a single block, a horizontal and a vertical block of two, and a block of four. We cannot rotate the blocks in our simple version.

In the setup() function we create a new stone:

```
function setup() {
    ...
    createNewBrick();
}
```

The createNewBrick() function creates a random new stone. The stone shape depends on a random number:

```
function 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);
}
```

The brick must be an global variable, otherwise it won't work. The same applies to our random generator. In order for the stones to start falling down, we need a game loop:

```
function draw() {
    moveBrick();
    checkForCollision();
    update();
}
```

The moveBrick() function simply moves the brick down by one brick width. The checkForCollision() function determines whether the stone is allowed to fall any further. Once the stone has arrived at the bottom, it just sits there. How do we do that? We simply create a new stone, and leave the old one where it is:

```
function checkForCollision() {
    // check bottom
    if (brick.getY() > HEIGHT - brick.getHeight()) {
```

```
            createNewBrick();
        }
        ...
```

However, we can also have collisions with other blocks, that are sitting already at the bottom. In this case the falling stone also is not allowed to fall any further. We need to use the getElementAt() function and check for collisions:

```
        // check for other bricks
            let x = brick.getX() + 1;
            let y = brick.getY() + brick.getHeight();
        let obj = getElementAt(x, y);
        if ((obj != null) && (obj != brick)) {
            createNewBrick();
            return;
        }
        ...
    }
```

This function tells us whether there is a GObject at the x,y position. If there is, we simply create a new stone.

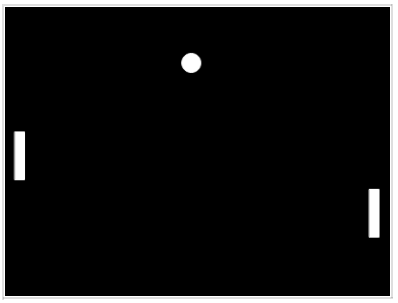
So, now our stones are falling. The only thing missing is key control. When a key on the keyboard is pressed, a KeyEvent occurs. This is completely analogous to the mouse events. Therefore, we need to add a keyPressed() function to our Tetris class:

```
    function keyPressed() {
        switch (keyCode) {
            case LEFT_ARROW:
                brick.move(-BRICK_SIZE, 0);
                break;
            case RIGHT_ARROW:
                brick.move(BRICK_SIZE, 0);
                break;
        }
    }
```

The *keyCode* tells us which key was pressed: for the left arrow key the keycode is 37, for the right the keycode is 39 and that is all we need to finish our simple Tetris game.

.

Pong



Try it

Pong was released by Atari in 1972 and is considered the forefather of all video games [8]. It is a game for two players who try to play a ball back and forth similar to ping-pong. To get started, we need three global variables for the ball and the two paddles.

```
    let ball;
    let leftPaddle;
    let rightPaddle;
```

Then of course there is our game loop:

```
    function draw() {
        moveBall();
        checkForCollision();
        update();
    }
```

The moveBall() function simply moves the ball by a certain amount in x and y direction.

The checkForCollision() function does two things: it checks if there was a collision with the wall or if there was a collision with one of the paddles. If the ball wants to leave the playing field on the top or bottom, it is simply reflected. If it wants to leave on the left or right, then it simply disappears, and the round is over. If the ball collides with the paddles, it is also reflected.

For the collisions with the paddles we use the getElementAt() function. Reflection is quite simple. We first need an global variable for the speed:

```
let vx = 2;
let vy = 3;
```

and then reflection simply means:

```
vy = -vy;
```

Of course, the moveBall() function must also use these variables.

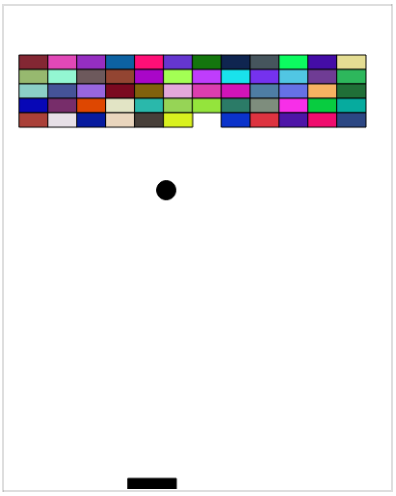
The paddles should be controlled by the keyboard, so we need to add the KeyListener again and add the keyPressed() function:

```
function keyPressed() {
    switch (key) {
        case 'p': // right paddle up
            rightPaddle.move(0, -PADDLE_VY);
            break;
        case 'l': // right paddle down
            ...
    }
}
```

Notice, we use *key* instead of *keyCode* here. We use *keyCode* if we want to detect special keys, however, for letters and digits, we use key. The first player controls her paddle with the keys 'q' and 'a', the second player with the keys 'p' and 'l'.

.

BrickBreaker



Try it

BrickBreaker is inspired by Breakout, a game that was introduced by Atari in April 1976. The game principle was developed by Nolan Bushnell. Steve Jobs, who then worked at Atari, persuaded his friend Steve Wozniak (then at HP) to write the game. Steve Jobs got paid $5,000 for the game by Nolan Bushnell. He gave his friend Steve Wozniak half, that is 350 dollars [12].

The playing field consists of bricks, a ball and a paddle. The ball moves through the playing field and when it hits a brick, the brick disappears. The ball is reflected from the walls and the paddle. Except for the lower wall: here the ball just vanishes and the game is over. In our game, the paddle is to be controlled by the mouse.

As global variables we need the ball and the paddle:

```
let paddle;
let ball;
```

In the setup() function, we initialize the ball and the paddle, and we also draw the wall of bricks:

```
function setup() {
  ...
  createBall();
```

```
  createPaddle();
  createBricks();
}
```

When drawing the wall, we can borrow our function drawRowOfNBricks(int n) from the Stairs project, so that is easy. Maybe we want to make the bricks differently colored, then the game looks much better.

After the setup we come back to the game loop:

```
function draw() {
  moveBall();
  checkForCollision();
  update();
}
```

The moveBall() function is identical to that in Pong, checkForCollision() is also very similar

```
function checkForCollision() {
  checkForCollisionWithWall();
  checkForCollisionWithPaddleOrBricks();
}
```

What needs a bit more work is the checkForCollisionWithPaddleOrBricks() function. We use the getElementAt() function again, and check if the object is not null. If it is not null, then the only options are, that either it is the paddle or a brick,

```
let obj = getElementAt(x, y);
if (obj !== undefined) {
if (obj === paddle) {
  vy = -vy;
} else if (obj instanceof GRect) {
  removeObj(obj);
  vy = -vy;
}
```

You may notice the *instanceof* keyword, that can be used to test if an object is a GRect, or any other type for that matter. Last thing we have to worry about, is the movement of the paddle through the mouse. For this we have to call the addMouseListeners() function in the setup() and we need to implement the mouseMoved() function:

```
function mouseMoved() {
  let x = mouseX;
  paddle.setLocation(x, HEIGHT - PADDLE_SEPARATION);
}
```

And that's it.

.

QUESTIONS

1. There are functions with a return value and some without. How can you tell that a function has no return value?

2. Some functions have parameters. What are parameters?

3. What do parameters and kilometers have in common?

4. We have used the random generator 'rgen' more than once. What is the command to generate a random

number (integer) between 1 and 6?

5. If you pass a primitive data type to a function as a parameter, is it passed in as the original or as a copy?

6. Name four subclasses (child classes) of the class GObject.

7. Draw a diagram that roughly represents the class hierarchy of the GObjects.

8. We learned about the top-down approach. This gives rules regarding the names of functions, how many lines of code a function should have, etc. Name two of these guidelines.

9. Analyze the game 'Agar.io' using the top-down approach. You only need to worry about the high-level structure, not the detailed code.

10. The addThirteen() function in the following code does not work as expected. What's the problem? How would you solve it?

```
function addThirteen( x ) {
    x += 12;
}
function run() {
    let x = 4;
    addThirteen( x );
    println( "x = " + x );
}
```

11. The arcade game 'RandomCircles' was an absolute hit in the late 1980s. It is a game that draws a colorful circle at the point where the user clicks with the mouse. It would be a little to much to ask to implement the full version of the game, but it is relatively easy to write the code that
    1) draws a colored circle on the screen and
    2) draws the circle where the user clicked with the mouse.

    The first step is to create a class called GCircle that extends GOval and is initialized with random size and color.

    The second step is to write a graphics program that listens to mouse clicks and draws a circle at the location where the user clicked with the mouse.

    The following information may be helpful:

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

    The RandomGenerator has among others the following functions:

    ```
    rgen.nextInt( low, high )
    rgen.nextColor()
    ```

.

REFERENCES

The references relevant for this chapter are the same as in Chapter 2. More details about the ACM graphics library can be found on the ACM Java Task Force [1] pages. Many more examples can be found in Tutorial [2], the book by Eric Roberts [3] and the Stanford Lecture 'Programming Methodologies' [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] 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

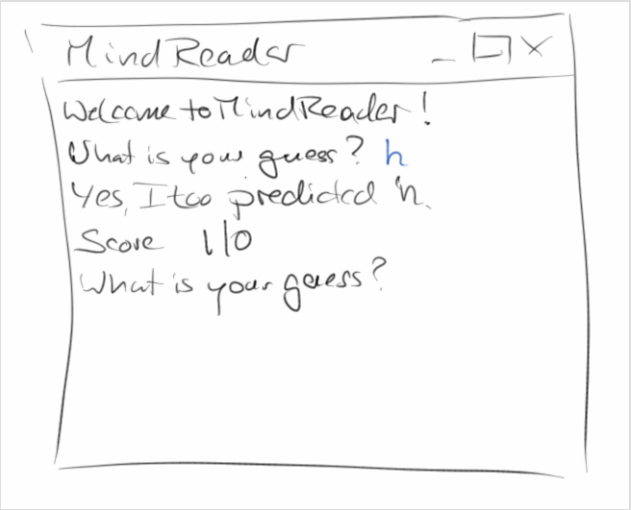
[10] Agar.io, https://de.wikipedia.org/wiki/Agar.io

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

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

.

.



MINDREADER

In this chapter we return to console programs. We will learn about text and word processing, that is how to use characters and strings. In addition, we will also deepen our understanding of classes. We will see how classes are structured and we will create our own classes.

.

String

Strings are made up of zero or more characters. A character can be a letters, a digits or a special character. The apostrophes (single or double quotes) are necessary to distinguish digits from numbers:

```
let x = 42;
let z = "42";
```

The first line declares the number 42, the second line the string "42". Special characters are characters like '.', '$', etc., but there are also invisible characters such as '\n' for a new line or '\t' for a tab.

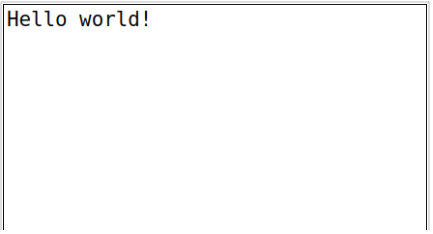
Let's continue with a few examples:

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

Here we declare two strings, s1 and s2, and initialize them to the values "Hello" and "world". Strings can contain none, one or many characters:

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

Here s3 contains an exclamation mark. s4 also contains a character, the space. s5 contains no character at all, but s5 is still a string.



What makes strings really interesting is that we can make new strings out of old ones by concatenating them:

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

Try it

and also print them

```
println( s6 );
```

And just as we can read numbers (int and double) from the console, we can also read strings from the console:

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

The String class has a few useful functions:

- **length:** returns the number of characters in a string,
- **charAt(pos):** returns the character at the position pos,
- **substring(p1):** returns the substring from position p1 to the end of the string,
- **substring(p1, p2):** returns the substring from position p1 to position p2, but without the character at position p2,
- **includes(s):** returns true if the string s is contained in the string,
- **indexOf(s):** determines if the string s is contained in the string, if it is it returns the position of the character, -1 otherwise,
- **trim():** removes white spaces at the beginning and end of a string, and
- **toUpperCase():** returns the uppercase version of a string, of course there is also a toLowerCase().

As we will see in a moment, there are quite a few things we can do with these functions.

.

Exercise: Print the Characters of a String

As a first example, we iterate through the characters of a string and display them one by one on the console:

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

.

Exercise: Reverse a String

We can use what we just learned to reverse a string:

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

The word "stressed" becomes the word "desserts". Words that remain the same when reversed, such as "racecar" or "otto", are called palindromes.

.

Exercise: Compare two Strings

Next, we want to determine whether two strings are equal. In JavaScript we can use the "==" operator for this:

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

As expected "Equal" is what gets prinited here. Just for curiosity, what happens if we compare numbers and strings:

```
let x = 42;
let y = "42";
if (x == y) {
    console.log("x and y are equal");
}
if (x === y) {
    console.log("x and y are exactly equal");
}
```

When we use the simple equality operator "==", JavaScript says they are equal. However, there is also the exactly equals operator "===" with three equal signs, and in that case they are not longer equal. The difference is that the exactly equal operator also checks if the two variables are of the same type.

SEP: Always use the exactly equals operator "===" when comparing variables.

.

Exercise: Cut a String

As a last little exercise, we want to cut a string into two pieces. For this we use the functions indexOf() and substring():

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

We find the position of the space using the indexOf() function. Then we cut off the front part with the substring() function. Two things are important here: the first character is at position 0, i.e. as usual we start counting at 0. And the character at the posSpace position is not included. Finally, we cut off the rest, and store it in s3.

.

StringTokenizer

Cutting strings into pieces is a very common task. That is why there is a special class for this, the StringTokenizer:

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

```
hi
there
what's
up
```

Try it

First, we initialize the StringTokenizer by telling it what it should cut apart (sentence), and how it should cut it. That means we tell it the separators it should use, like ' ', '.', ',' or '?', for instance. The StringTokenizer then splits the string into tokens, i.e., words. To get to these words, we ask the StringTokenizer if it has any tokens, so we call the

hasMoreTokens() function. If it has any, we ask the StringTokenizer to give us the next token using nextToken(). We do this until all the tokens are used up.

.

Immutability

Something that sometimes leads to confusion is the immutability of strings. Immutability means that a string cannot be changed. Let's take a look at an example:

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

```
Hello
HELLO
Equal
More Equal
```

Try it

What is displayed on the console? "Hello" is displayed, and not, as one might hope, "HELLO". This has to do with the fact that strings are unchangeable, i.e. immutable.

But there is a trick to changing strings anyways: we simply assign a new value to them:

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

.

Classes

We have already heard a lot about classes, and we have used them many times, e.g. the Karel class or the graphic classes like GRect and GOval. Also, in this chapter we used the String class and the StringTokenizer class. The question that arises is: can we make our own classes?

Of course, and it's not that hard. As a first example, we look at a student from the perspective of a university. What information would a university need to know about a student? If we limit ourselves to the essentials, then these are name, matriculation number and credit points (ECTS). We take these attributes of a student and combine them into a class:

```
class Student {
    constructor(_name, _id, _credits) {
        this.name = _name;
        this.id = _id;
        this.credits = _credits;
    }
}
```

In principle, a class is nothing more than a data container. Each class should be saved in its own file, the name of the file should be the same as the name of the class with a '.js' extension.

SEP: Class names should always start with capital letters and follow the CamelCase convention.

.

Constructor

Classes are 'intelligent' data containers. This means that they not only contain and bundle the data, but also initialize, manage and change it. So they also do something with the data, and classes have functions for that. Usually, we call functions that belong to classes *methods*, to distinguish them from the normal functions. The most important method is the constructor, which initializes the data.

```
class Student {
    constructor(_name, _id, _credits) {
        this.name = _name;
```

```
            this.id = _id;
            this.credits = _credits;
        }
    }
```

The constructor, like any function, can have parameters. A constructor never returns a value, because it is implicitely assumed, what it returns is an *instance* of that class.

Now we have a class, how do we use it? We use them like any other class. Let's write a console program which creates an instance of our Student class:

```
async function setup() {
    ...
    let fritz = new Student("Fritz", 12345, 0.0);
        println(fritz.name + ", " + fritz.id + ", " + fritz.credits);
}
```

We create a new instance of the class Student. This instance (also called object) we call fritz. To create the object we have to call the constructor with "new". We initialize the instance variables of the class: that is, name is initialized to the value "Fritz", id to the value 12345, and credits to the value 0.0.

.

Methods

Let's say we want to print this information about fritz quite often. Then every time, we would have to write the above print statement. However, if we create a method called toString(),

```
class Student {
        ...
    toString() {
        return "Student [name=" + this.name + ", id=" + this.id
               + ", credits=" + this.credits + "]";
    }
}
```

then printing is much easier:

```
async function setup() {
    ...
        println(fritz.toString());
}
```

Another example is incrementing credits. Fritz has been studying hard, passed his Programming exam, and now he wants he credits updated:

```
            fritz.credits = fritz.credits + 5;
```

More convenient would be an incrementCredits() method:

```
            fritz.incrementCredits(5);
```

This is more for convenience, but makes our code more readable.

SEP: Every class should have a toString() method.

.

Closures

Now notice, we can read fritz's name, but we can also change it:

```
println(fritz.name);
fritz.name = "Hans";
println(fritz.name);
```

JavaScript is pretty open about accessing the instance variables of a class. Sometimes you are fine with that, but sometimes you don't want this.

Consider the following class, where we added a method named *getId()*:

```
class Student {

    constructor(_name, _id, _credits) {
        this.name = _name;
        this.credits = _credits;

        // closure: read only
        this.getId = (function () {
            let id = _id;
            return function () {
                return id
            }
        })();
    }
}
```

It looks very strange, but it does what we wanted: we can read the *id*, but we can no longer change it:

```
    let fritz = new Student("Fritz", 12345, 0.0);
        println(fritz.getId());  // 12345
    fritz.id = 42;
    println(fritz.getId());  // 12345
```

JavaScript just ignores the assignment. Closures are kind of complicated, and we will not use them often. But you know now, that they exist.

.

Exercise: Hansel und Gretel

The class Student is now ready for use and as an example we take a look at the two students Hansel and Gretel at the Brothers Grimm college:

```
async function setup() {
    createConsole();

    let hansel = new Student("Hänschen", 12345, 0.0);
    println(hansel.name);

    let gretel = new Student("Gretel", 54321, 11.0);
    println(gretel.name);
    gretel.name = "Gretchen";
    gretel.incrementCredits(5);
    println(gretel.toString());
}
```

```
Hänschen
Gretel
Student [name=Gretchen, id=54321,
credits=16]
```

Try it

.

Type Conversion

JavaScript has a few functions for converting between strings and numbers. It is not uncommon that we have to convert a string into a number.

```
let x = parseInt("42");
let y = parseFloat("42.0");
```

Sometimes we also want to convert a floating point number into an integer number:

```
let z = Math.trunc(42.0);
```

Conversely, to create a string from a number, you can use a little trick:

```
let x = 42;
let fourtyTwo = "" + x;
```

There is also other ways, but knowing one is fine.

.

REVIEW

In this chapter we have mainly dealt with strings and classes. Also, it is the first time we've written our own class. We have seen that a class has variables and methods. Sometimes we also call the variables instance variables, class properties, or class attributes. We have seen that methods always do something, and that there is a special method, the constructor. We also learned about the String and StringTokenizer classes.

.

PROJECTS

The projects in this chapter mainly deal with strings and writing simple classes.

.

Student

In the example for Student the question arises why are the instance variables private? Let us look at the problem from the other side. Suppose instance variables were public: could we then prevent a student from getting negative credits, or that her id could be changed?

.

ASCII

To store characters internally the computer uses numbers. This means we can add and subtract these characters just like numbers, and we can even use them in conditions and loops. For example, to determine if a character is an uppercase letter, you can use the following condition:

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

Or to turn a lowercase letter into an uppercase letter, we can use the following trick:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~¨
```

Try it

```
let klein = 'd';
let gross = klein.toUpperCase();
```

You can also make ints out of chars and vice versa via a cast, that is a type conversion

```
let char = String.fromCharCode(i);
print(char);
```

With this knowledge we want to write the following three functions:

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

As for the latter, it is sufficient to output the ASCII characters between 32 and 128, since the first 31 ASCII characters cannot be printed.

.

PasswordCreator

With our knowledge of strings we can now write a password generator. A good password should be at least 8 characters long and contain at least one lowercase letter, one uppercase letter, one digit and one symbol. So we could define four strings containing the possible characters, and then use the RandomGenerator to randomly select two of them and put them together to a password, for example:

```
bV9(wD6;
```

Try it

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

.

RandomText

For test purposes it is sometimes helpful to be able to generate random text. That's not so hard. We start again with the top-down approach. Let's assume we have a function createRandomSentence() that generates a random sentence. Then we could create a random text by simply creating several random sentences. The createRandomSentence() function, in turn, is easy if we assume there is a function called createRandomWord(), because then we would just take somewhere between three and five words, put spaces between them and add a period at the end. Also the createRandomWord() function is not that difficult, because words consist of lower case letters, and have a length of about 3 to 8 letters, so

```
cmdhitz ntms orja utrew vqupc.
```

Try it

```
function createRandomWord() {
    let word = "";
    let nrOfCharsInWord = rgen.nextInt(3, 8);
    for (let i = 0; i < nrOfCharsInWord; i++) {
        word += String.fromCharCode('a'.charCodeAt(0) + rgen.nextInt(26));
    }
    return word;
}
```

.

Palindrome

Words that remain the same when reversed, such as "racecar" or "pensioner", are also called palindromes. We want to write a function called isPalindrome(String s), which determines if a string is a palindrome. To do this, we write a function reverse() that reverses a given string, and then compare the original with the reversed string using equals(). If both are equal, it is a palindrome.

```
Enter String: racecar
'racecar' is a palindrome.
```

Try it

.

.

CountUpperCase

Here we should write a function that counts the uppercase letters in a string. This can be very useful, for example, if you want to determine whether a certain text is German or English: English text has on average less capital letters per sentence. We can either use the character class or the function we wrote above.

```
Enter string: Hello World
The string has 2 upper case chars.
```

Try it

.

.

.

ReadOneChar

We want the user to be able to enter only one letter. The user should not be able to enter zero or more than one letter, but only exactly one letter. Therefore we want to write a function readOneChar(), which returns one char as return value.

There is the readString() function for reading strings. Hence, we use the loop-and-a-half, and use as abort criterion that the string read should be of length one.

```
Enter Y or N: Yes
You idiot, you must enter only one
character:
Enter Y or N: Y
You entered: Y
```

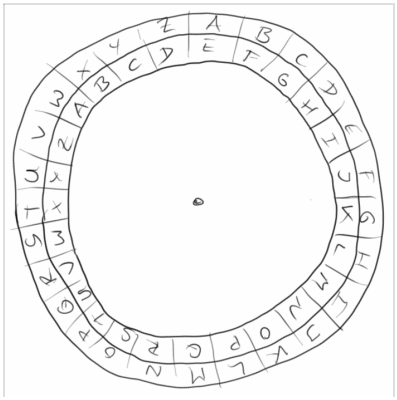
Try it

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

.

Encrypt

Another interesting application that we can now easily implement is a small encryption program. We use the so-called Caesar cipher, named after Julius Caesar [2]. To encrypt a text using Caesar's cipher, the letters in a given text are simply shifted by a fixed number, the key. For example, if the key is four, then an 'a' becomes an 'e', a 'b' becomes an 'f', etc. To decrypt, the process is simply reversed.



To implement this in code, we write two functions, encrypt() and decrypt(), one for encrypting, the other for decrypting. Both take two parameters, a string and an int, the key. One returns the encrypted text as a string, the other returns the decrypted text.

.

.

.

Note: in order not to make it too complicated, it makes sense to convert the text to lower case before starting the encryption. And the remainder operator '%' is very practical here:

```
Enter String: HelloWorld
khoorzruog
helloworld
```

Try it

```
function encryptChar(c, key) {
    let d = c.charCodeAt(0) - 'a'.charCodeAt(0);
    let e = d + key;
    let f = e % 26;
    let g = String.fromCharCode(f + 'a'.charCodeAt(0));
    return g;
}
```

.

Abjad

Around 1500 B.C. the Phoenicians developed the first alphabet script, a left-hand consonant script. Consonant scripts are writing systems in which only consonants are being used [2].

To show that such a script actually can be readable, archaeologists have commissioned us to write a console program that removes all vowels from a given text. We can proceed as follows:

```
Enter text: Hi there, how are you?
H thr, hw r y?
```

Try it

- with readLine() we ask the user to enter a normal text,
- we then look for vowels and remove them, and
- we then output the result using println().

Ideally we should remember the top-down approach, i.e. we should perhaps assume that the functions removeVowels() and isVowel() exist, and then implement them later. You could also use the replace() function of the String class, but more about this in the next project.

.

Franconian

The language of the Franks, also called 'lingua franca', has been wrongly forgotten. The 'lingua franca' (Italian for 'Franconian language') is a Romance-based pidgin language. Pidgin language or pidgin refers to a reduced form of language that is used by people of different languages to communicate [3].

```
Enter German text: Politiker
bolidiger
```

Try it

In order to contribute to the understanding of peoples, we should write a German-Franconian translation program. In Franconian the following phonetic simplifications take place (also known as "inner-German weakening of consonants"):

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

For example, the word 'Politiker' (politician) becomes 'Bolidiger' in Franconian.

So let us write a function translateGermanToFranconian(), which has a string as parameter with the German text, and returns a string, which is the translated Franconian version. For this you could use the replaceAll() function of the String class:

```
String german = "politiker";
```

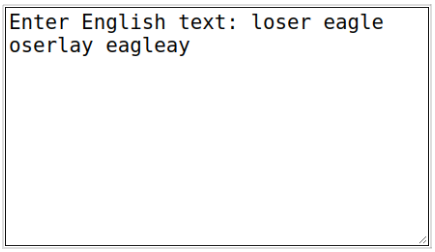
```
String franconian = german.replaceAll('t','d');
...
```

But replaceAll() and also replace() use "regular expressions". So unless you know what those are, you should waiting using them until your second semester. However, a function like the following is all you need:

```
function translateChar(c) {
    switch (c) {
        case 't':
            return 'd';
        case 'k':
            return 'g';
        case 'p':
            return 'b';
        default:
            return c;
    }
}
```

.

PigLatin



Try it

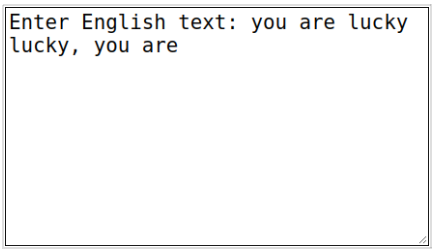
Pig Latin [5] is a secret language for children that has very simple rules:

- if a word begins with a consonant, then the consonant is moved to the end of the word and the syllable "ay" is appended. So "loser" becomes "oserlay" or "button" becomes "uttonbay",
- if a word begins with a vowel, then only the syllable "ay" is appended. So "eagle" becomes "eagleay" and "america" becomes "amercaay".

Hence, we should write another console program that asks the user for an English sentence and then translates it into Pig Latin. Here, too, we should use of the top-down approach.

.

YodaTalk



Try it

In the typology of languages [6], the sentence structure subject-verb-object (SVO), i.e. the subject is in the first place, the verb in the second and the object in the third, occurs very frequently. About 75% of all languages in the world follow this or a very similar pattern. We know that Yoda comes from the swamp planet Dagobah and there the preferred sentence construction is object, subject verb (O,SV).

For example, the English sentence,

> You are lucky.

translates into "Yodish":

> Lucky, you are.

To simplify interplanetary communication, it is now our task to write a console program that translates a given text from English into Yodish. We can assume that each sentence entered always consists of three words, always in the order SVO.

Here's what we could do:

- we ask the user to enter an English text,

- then we identify the three elements subject, verb and object using a StringTokenizer,
- and using println(), we then output the translated Yodish in the form object, subject verb.

Of course we use the top-down approach again and implement a function called translateFromEnglishToYodish().

.

RandomGenerator

We want to write our own RandomGenerator class. This is not so difficult when we know that there is a function called Math.random() in standard JavaScript. This function returns a floating point number between 0 and 1, including 0, but excluding 1. If we want to use it to generate a number between 1 and 6, this is how it works:

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

```
2,9,8,5,6,0,6,1,1,5,
1,1,2,5,1,2,2,5,1,3,
true,true,true,true,true,true,tru
e,false,false,true,
rgb(7,24,11)
rgb(16,8,205)
rgb(168,38,147)
rgb(143,26,169)
rgb(210,180,19)
rgb(228,1,81)
```

Try it

With this knowledge we want to write a class called RandomGenerator, that has the methods nextInt(int a, int b), nextInt(int b), nextBoolean() and nextColor(). In the future we can always use our own class instead of the ACM class, if we want to.

.

Counter

We want to create a class Counter that can act as a counter. The class should have a private instance variable called count. In the constructor, this variable is to be set to the value 0. Then there should be a method called getValue(), which simply returns the current value of the variable count. And there should be a method called incrementCounter(), which increases the value of the variable count by one. Can you prevent somebody else from messing with your counter? (Closure)

```
Value of cntr1: 1
Value of cntr2: 3
Value of cntr1: 1
```

Try it

.

.

Point

Next we want to write a class Point. This should correspond to a point in two-dimensional space, i.e. have an x and a y coordinate. The class should have a constructor of the form Point(int x, int y), and the methods: getX(), getY(), move(int dx, int dy), equals(Point p), add(Point p), and toString().

```
p1: Point [x=2, y=3]
p2: Point [x=-1, y=1]
p3: Point [x=3, y=2]
p3+p2 == p1?: true
```

Try it

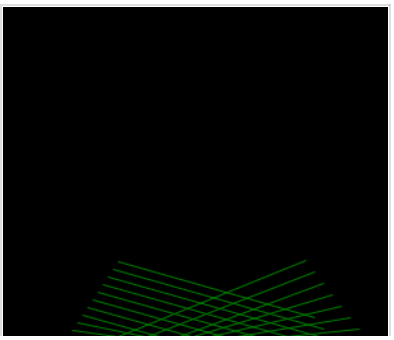
.

.

.

ConnectTwoMovingPoints

Our class Point seems relatively boring on its own. But if we use it for the motion of two points which are connected by a colored line, then this actually looks quite nice.



We start with the Point class, which, however, has slightly different requirements than the one of the last project. It should be as simple as possible. It should have four instance variables, namely x, y, vx and vy. The constructor should have no parameters, and it should initialize the instance variables with some small random values. This Point class should have only one method:

```
    move() {
        this.x += this.vx;
        this.y += this.vy;
    }
```

Try it

We want to use this class in our graphics program ConnectTwoMovingPoints:

```
let p1;
let p2;

function setup() {
    ...
    p1 = new Point();
    p2 = new Point();
}

function draw() {
    p1.move();
    p2.move();
    checkForCollisionWithWall(p1);
    checkForCollisionWithWall(p2);
    drawConnection(p1, p2);
    update();
}
```

In our game loop, we move the two points, check if the points are still in the field, and then connect the two points with a line. That's actually quite simple. However, if you only want to display ten lines at a time and want to delete the older lines again, you need arrays (next chapter).

.

Blackjack

```
Your current score is: 2
Do you want another card (Y/N)? Y
Your current score is: 7
Do you want another card (Y/N)? Y
Your current score is: 12
Do you want another card (Y/N)? Y
Your current score is: 21
Do you want another card (Y/N)? N
you: 21 | house: 17
You win.
```

Try it

According to Wikipedia, "Blackjack is the most widely played card game offered in casinos" [9]. We want to implement a slightly simpler version of this game.

Instead of cards, we simply use numbers, numbers between 1 and 11. The computer plays the dealer and starts by generating a random number between 17 and 25. Then it's the player's turn. She starts with a card, i.e. a random number between 1 and 11, and can then decide whether she wants another card. If yes, another random number between 1 and 11 is generated and added to the current "hand". When the player no longer wishes to have a new card, the "hand" of the player is compared with that of the computer.

The winner is whoever has 21 or less points and has more than the other player. Otherwise, it's a draw.

.

SimpleCraps

```
You have €100.
Enter your bet: odd
The dice shows 1.
The number is odd, you win.
You have €110.
Enter your bet: high
The dice shows 5.
The number is high, you win.
You have €120.
Enter your bet:
```

Try it

"Craps or Seven Eleven is a dice game that enjoys great popularity, especially in the USA." [10]

We will implement a simpler version of Craps: We have just a simple dice. The player starts with credits of 100 Euro. In each round, 10 Euro are bet and the player bets on one of the following outcomes:

- odd
- even
- high (4,5,6)

- low (1,2,3)

The game is over when the player has no more credits.

.

Factorial

The factorial of a number is the product of all integer numbers less than or equal to that number. For example, the factorial of 3 is:

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

We want to write a function called calculateFactorial(int n) which calculates the factorial of the number n. With that function we then want to list the factorials of the numbers from 1 to 20. Probably using a for-loop is a good idea.

The factorials have the property that they become very large very quickly. And that leads to a problem, because computers do have some problems, when it comes to large numbers! What does JavaScript say when the numbers get to large?

```
Factorial 0 is: 1
Factorial 1 is: 1
Factorial 2 is: 2
Factorial 3 is: 6
Factorial 4 is: 24
Factorial 5 is: 120
Factorial 6 is: 720
Factorial 7 is: 5040
Factorial 8 is: 40320
Factorial 9 is: 362880
Factorial 10 is: 3628800
Factorial 11 is: 39916800
Factorial 12 is: 479001600
Factorial 13 is: 6227020800
Factorial 14 is: 87178291200
Factorial 15 is: 1307674368000
Factorial 16 is: 20922789888000
Factorial 17 is: 355687428096000
Factorial 18 is: 6402373705728000
Factorial 19 is:
```

Try it

.

.

.

Rabbits

Anyone who has ever had rabbits knows that they have an interesting characteristic: they reproduce rapidly (that's why they are called rabbits). To see how rapidly, let us write a program that calculates how our rabbit population develops over the months. We follow the model of Fibonacci [12]:

```
1,1,2,3,5,8,13,21,34,55,89,144,
```

Try it

- "Each pair of rabbits throws another pair of rabbits every month.
- A newborn couple only has offspring in the second month of its life (the gestation period of rabbits is about one month).
- All rabbits are in an enclosed space so that no animal can leave the population and none can come in from the outside."

If we write the simulation correctly, then the Fibonacci sequence, i.e. the numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... should come out.

.

WordTwist

The WordTwist game is about recognizing a word in which some letters have been swapped. So, for example, we should realize that the word "nhickce" actually originated from the word "chicken".

```
odg
Your guess: dog
correct
cehikcn
Your guess: chicken
correct
act
Your guess:
```

Try it

We begin with the starting word, e.g. "chicken", and swap some letters. We show this word to the user, and he should then guess which was the original word. If we use the top-down approach again (as we always should). Then it makes sense to write a function scrambleWord(String word) that takes the starting word as parameter and returns the scrambled word as return value. If we continue the top-down approach, it also makes sense to write a function called randomSwap(String word) that simply swaps two random letters in the

word. If we call this function several times, the result is a well-shuffled word. What we still need is a function that picks a random word to start with:

```
function pickGuessWord() {
    let rgen = new RandomGenerator();
    switch (rgen.nextInt(0, 3)) {
    case 0:
        return "dog";
    case 1:
        return "cat";
    default:
        return "chicken";
    }
}
```

.

CHALLENGES

.

Hangman

Hangman is a simple letter game [12]. Although it is a graphical game originally, we will implement a text-based version here. It starts with the computer picking a random word. The player may then guess one letter at a time. The program then shows whether this letter occurs in the word and if it does, at which position it occurs. This is then repeated until the word has been guessed. At the end the user should see how many attempts were necessary to guess the word. However, only failed attempts are counted.

```
Welcome to Hangman
---
Guess a character: o
-o-
Guess a character: d
do-
Guess a character: g
dog
It took you 3 guesses
```

Try it

We need two instance variables,

```
let guessWord;
let hintWord;
```

where guessWord contains the word to be guessed (e.g. "mother") and hintWord contains the letters that were guessed correctly so far, in the beginning it contains only dashes (e.g. "------"). We need to initialize both guessWord and hintWord:

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

where the function pickGuessWord() already exists (see WordTwist above) and we still have to write the function createHintWord().

After that the game loop starts: In the first step we show the player the hintWord so that she has an idea how many letters the word contains. Then we ask the player to enter a letter. Here we can use the function readOneChar() from the project ReadOneChar. By means of *includes()*,

```
let c = await readChar();
if (guessWord.includes("" + c)) {
    buildNewHintWord(c);
}
```

we can determine if the new letter appears in the guessWord. If yes, we must construct a new hintWord using the buildNewHintWord(c) function. This function should insert the correct letter in the correct place in hintWord, and the

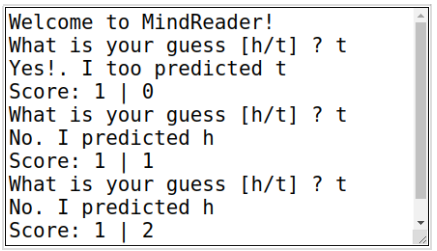
new hintWord should then be displayed to the player.

The abort criterion is relatively simple, if there are no more dashes "-" in the hintWord, then the player has guessed the word. Alternatively we could also compare guessWord and hintWord, if they are equal the game is over.

.

MindReader

This game is about mind reading. It is inspired by a coin toss where we have heads and tails. But instead of tossing a coin, the player simply picks either heads or tails. The computer now tries to predict what the player guessed. If the prediction of the computer was correct, the computer receives one point, otherwise the player gets one. Whoever gets 25 points first, wins. The idea of the game is based on a handout by Professor Raja Sooriamurthi, who in turn was inspired by Professor Gregory Rawlins [13].



Try it

If we use the top-down approach, we get the following rough structure for the game:

- the computer makes a prediction, either head ('h') or tail ('t') (computerMakePrediction()),
- the player makes his choice, either head or tail (humanMakePick()),
- the prediction of the computer is displayed (revealPrediction()),
- the computers prediction is compared with the selection of the player and depending on whether the computer has guessed correctly or not, either the computer gets a point or the player.

The whole thing is then repeated until either one has reached 25 points.

For the game we need four instance variables:

```
let computerGuess;
let humanGuess;
let computerScore = 0;
let humanScore = 0;
let predictor;
```

The character entered by the player, the character predicted by the computer, as well as the scores, i.e. the player's score and the computer's score.

We could write our own predictor (see extensions), or we can use the one from Professor Sooriamurthi:

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

For this to work we must include the MindReaderPredictor.js file at the beginning of our code with the line:

```
include("Pr5_MindReader/mindReaderPredictor.js");
```

The class MindReaderPredictor has two functions, makePrediction() and addNewGuess(char c). The first function tries to make a prediction, so it returns either 'h' or 't'. The second adds a new player guess to the Predictor database, allowing the Predictor to make better predictions. So the Predictor tries to learn from the player.

Extensions: One can come up with many extensions for this game. e.g.:

- you could play several games in a row, but MindReaderPredictor should not be reinitialized each time, or
- you could write your own MindReaderPredictor class, a very simple version would just randomly chooses between 'h' and 't'.

.

.

QUESTIONS

1. Give an example of a class and an example of an object.

2. "arnold" is an "actor". Is "arnold" an object or a class?

3. What is usually the job of a constructor?

4. It is good style to give each class a toString() method. What should the toString() method do?

5. How do you convert a string containing a number, e.g. "42", into an integer number?

6. Strings are immutable, what does that mean?

7. If you want to compare strings and numbers you have to be a little careful. What is the difference between the "==" and the "===" operators?

8. The class String has many methods. We used the following:
   substring()
   length()
   charAt()
   toLowerCase()
   indexOf()
   Briefly describe what each of these methods does.

9. Write example code that reverses a string, e.g. converts the word "STRESSED" into the word "DESSERT". (The pure JavaScript code is sufficient, a class or method declaration is not needed).

10. What is the StringTokenizer class used for?

11. The class StringTokenizer takes a string as parameter and has two methods called hasMoreTokens() and nextToken(). Write code that asks the user for a sentence, and then outputs the individual words of the sentence, line by line.

.

REFERENCES

The references from Chapter 2 are still important here. The Wikipedia also contains a lot of information.

[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

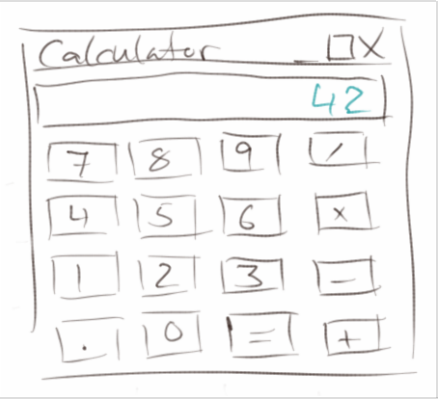
[11] Fibonacci-Folge, https://de.wikipedia.org/wiki/Fibonacci-Folge

[12] Galgenmännchen, https://de.wikipedia.org/wiki/Galgenmännchen

[13] Mind Reader: a program that predicts choices, http://nifty.stanford.edu/2007/raja-mindreader/

.

.



SWING

Programs with graphical user interfaces (GUI) are the topic of this chapter. Similar to graphic programs, there are ready-made components with which we can build small and large programs. It's a bit like Lego. In addition, we will learn some more about instance variables.

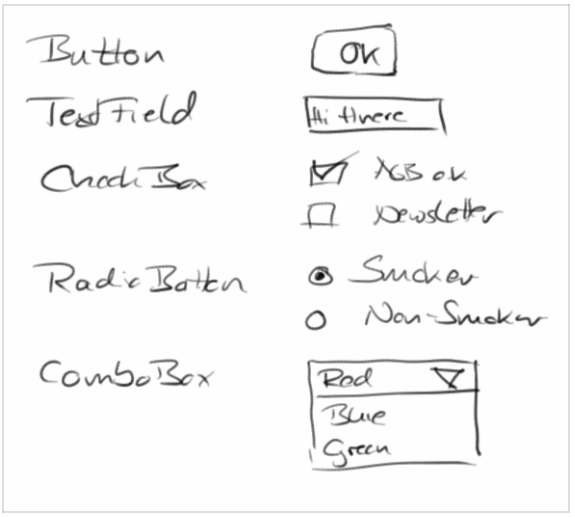
.

Graphical User Interface

Most of the programs we deal with every day are programs with a graphical user interface (GUI or UI). GUIs consist of graphical elements, which we also call widgets or interactors, because we can interact with them. Well-known examples are:

- Labels
- Buttons
- Textfields
- Checkboxes
- Radiobuttons
- Comboboxes

In the following we will see how to use them.

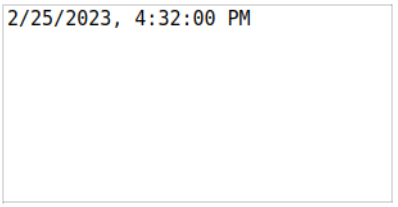


.

.

JSLabel

We start again very simple, with the JSLabel. There is practically no difference to the Glabel we already know. It is used to display text in GUI programs:



Try it

```
function setup() {
    createGUI(300, 150);

    let dt = new Date();
    print(dt.toLocaleString());
    let fritz = new JSLabel(dt.toLocaleString());
    fritz.addStyle('font: 15px monospace;');
    addWidget(fritz);
}
```

We see two differences to our previous programs: we now write createGUI() instead of createCanvas(). And we use

mostly the setup() function, the draw() we will hardly use. Furthermore we see for the first time the class Date. This is very practical when you need date or time. If you are familiar with CSS, then the content of the addStyle() method should look familiar to you.

.

JSButton

The JSLabel is not very interactive, and doesn't really deserve to be called an 'interactor'. That is different for the JButton, because you can click on it with the mouse. We'll start again very simple:

Try it

```
function setup() {
    createGUI(300, 150);
    setLayout('border');

    let btn = new JSButton("OK");
    addWidget(btn, 'SOUTH');
}
```

We create a new JSButton, on which "OK" should be written. We'll add that to the south of our program. When we start the program, we can press the button, but nothing happens.

For the button to work properly, that is, to become interactive, we have to implement the actionPerformed() function. It is like the Hollywood principle: don't call us, we'll call you. This is like with mouse events, if you remember. But instead of the mousePressed(), now the function actionPerformed() is being called. This means that every time someone presses the button, this function is called.

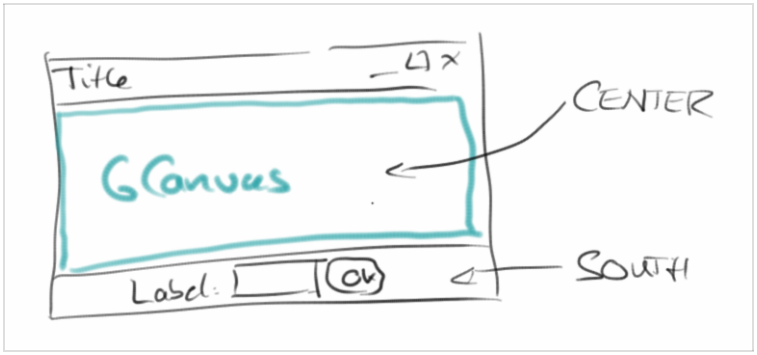
```
function actionPerformed(ev) {
    print("hi:" + ev.getActionCommand());
}
```

Now every time we press on the button with the mouse, in the console part of our program "hi:" together with the buttons name is being displayed.

.

Regions

In the two examples above, we have already used regions, NORTH and SOUTH. In total, there are three regions: SOUTH, NORTH, and CENTER. We can insert our widgets into any one of these regions.



.

JSTextField

Using the JSTextField widget we can read text and numbers. In the following example, we want the users to log in with their names:

Try it

```
let tf;

function setup() {
    createGUI(300, 150);
    setLayout('border');

    let lbl = new JSLabel("Name: ");
```

```
    addWidget(lbl, 'SOUTH');
    tf = new JSTextField(10);
    addWidget(tf, 'SOUTH');
    let btn = new JSButton("Login");
    addWidget(btn, 'SOUTH');
}

function actionPerformed(ev) {
    print("Name: " + tf.getText());
}
```

We create a JSLabel, a JSTextField and a JSButton. The constructor of the JSTextField needs to know how wide the field should be. It is also important that the JSTextField variable *tf* is a global variable. If it were a local variable, then we could not access it in the actionPerformed() function.

.

Global Variables

We have mentioned global variables briefly in the last chapters. The above example nicely shows what global variables are actually good for: they allow information to be exchanged between different functions. Up to now we could do this only by using parameters and return values.

Another reason why global variables can be very practical is that all local variables are deleted when a function is exited. For example, in the rollTheDie() function,

```
function rollTheDie() {
    let rgen = new MyRandomGenerator();
    let dieRoll = rgen.nextInt(1,6);
    println(dieRoll);
}
```

a new random generator is created each time the function is called and then it is deleted again when the function is exited. Both take time. But if we create the RandomGenerator as a global variable,

```
let rgen = new MyRandomGenerator();

function rollTheDie() {
    let dieRoll = rgen.nextInt(1,6);
    println(dieRoll);
}
```

it is created only once, and in addition we can use it in any function of our class. This is much more resource-saving.

.

Functions and Methods

We have been using the words function and method, and it maybe a little confusing what the difference is. It is actually very easy:

```
async function setup() {
    ...
    let hansel = new Student("Hänschen", 12345, 0.0);
    println(hansel.name);
    ...
}

function rollTheDie() {
    ...
}
```

```
class Student {

    constructor(_name, _id, _credits) {
        ...
    }

    incrementCredits(_credits) {
        ...
    }

    toString() {
        ...
    }
}
```

When a function belongs to a class, it is called a method. In the above example, constructor(), incrementCredits() and toString() are methods, because they belong to the class Student. Also in Math.random() or GRect.setColor(), both random() and setColor() are methods, because they belong to a class.

Conversely, setup() and rollTheDie() are functions, because they do not belong to any class. Also, println() and readLine() are functions, they also do not belong to any class.

.

Global Variables vs Instance Variables vs. Locale Variables

Confucing may also be the difference between the three types of variables we have been using: global, instance and local variables. Consider this example:

```
let rgen = new MyRandomGenerator();

async function setup() {
    ...
    let hansel = new Student("Hänschen", 12345, 0.0);
    ...
}

class Student {

    constructor(_name, _id, _credits) {
        this.name = _name;
        ...
                let local = 5;
    }
}
```

Global variables are declared outside of any functions, like *rgen* above. They can be accessed from anywhere, and they live forever.

Instance variables are declared in the constructor of a class and and always belong to a class or an object, like *name* in the above example. They are visible inside the entire class and all its methods, and they live as long as the object they belong to exists.

Conversely, local variables are declared within a method or function, can only be accessed within this function, and are deleted when leaving the function. In the above example *hansel* is a local variable, as is *local*. Also parameters to functions or methods are considered to be local variables.

Now the question arises, when do we use which? The answer is relatively simple in most cases:

- if it is a calculation that can be performed locally in a method, then local variables are used, e.g. the conversion from degrees to fahrenheit is a local calculation,
- if an information is used in several methods, we should use an instance variable, if all those methods belong

to the same class.

- if we are dealing with an internal state of an object, we should use an instance variable. For example, in the Student class, the name, ID and credits were internal states.

SEP: If possible, use local variables.

.

Exercise: OKCancel

As a little exercise we add a cancel button to our program. How can we tell which of the two buttons was pressed? One way to do this is to use the getSource() method of the ActionEvent,

Try it

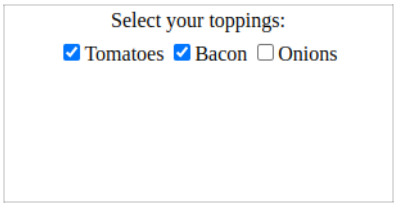
```
let btn1 = new JSButton("OK");
...
function actionPerformed(ev) {
    if (ev.getSource() === btn1) {
        print("source: btn1");
    } else if (ev.getSource() === btn2) {
        print("source: btn2");
    } else {
        print("unknown source");
    }
}
```

to distinguish between several buttons or widgets in general. In this case the buttons or widgets must be instance variables.

.

JSCheckBox

Next we will consider the JSCheckBox widget: we use check boxes when we have several choices.



Try it

```
let topping1;

function setup() {
    ...
    topping1 = new JSCheckBox("Tomatoes");
    addWidget(topping1, 'CENTER');
    let topping2 = new JSCheckBox("Bacon");
    addWidget(topping2, 'CENTER');
    let topping3 = new JSCheckBox("Onions");
    addWidget(topping3, 'CENTER');
}
```

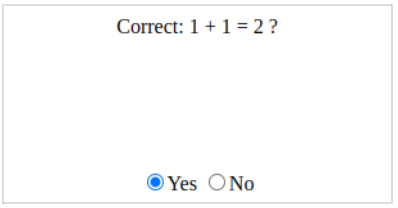
In the pizza example, we want to be able to select all possible combinations of toppings. This can best be achieved with check boxes. To determine which toppings were selected, there is the method isSelected():

```
function actionPerformed(ev) {
    print("source: " + ev.getSource().isSelected());
    if (ev.getSource() === topping1) {
        print("Tomatoes:" + topping1.isSelected());
    }
}
```

We have to do that for all three toppings, of course.

.

JSRadioButton



Try it

Radio buttons are used when making decisions. Let's look at the following example:

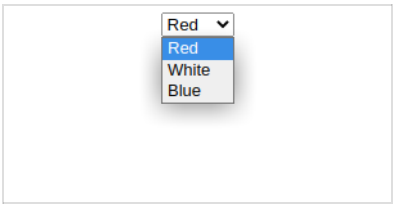
```
let yes;
let no;

function setup() {
    ...
    yes = new JSRadioButton("Yes");
    yes.setSelected(true);
    addWidget(yes, 'SOUTH');

    no = new JSRadioButton("No");
    addWidget(no, 'SOUTH');
}
```

Here we have two radio buttons, where the "yes" button is preselected. Radio buttons belong together in the sence that only one of the buttons can be selected at a time. If you want to know which button the user has selected, you can do this using:

```
let b = yes.isSelected();
```

.

JSComboBox



Try it

As an example for the JSComboBox we look at the example FavoriteColor. It is about selecting a color from a list:

```
let colorPicker;

function setup() {
    ...
        colorPicker = new JSComboBox();
    colorPicker.addItem("Red");
    colorPicker.addItem("White");
    colorPicker.addItem("Blue");
    addWidget(colorPicker, 'NORTH');
}

function actionPerformed(ev) {
    println("Color:" + colorPicker.getSelectedItem());
}
```

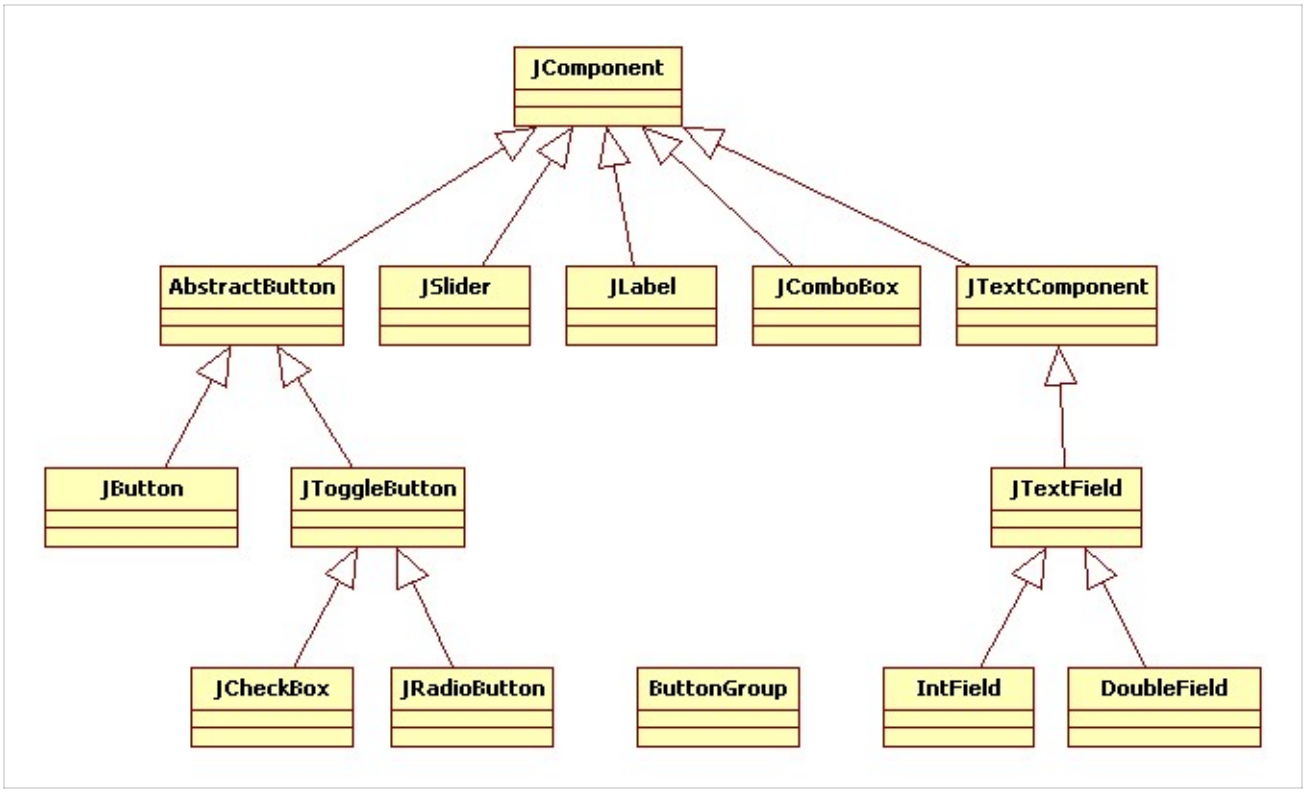
In the init() method we initialize the colorPicker with the preset colors. To determine which color the user has selected, there is the getSelectedItem() method. But to access them in the actionPerformed() method, colorPicker must be an instance variable.

Note: if we closely observe the behavior of the JSComboBox, we will notice that it has some small quirks. There's nothing we can do about it.

.

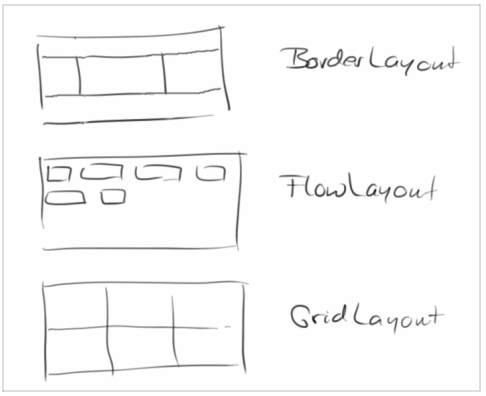
UI Interactor Hierarchy

Similar to the hierarchy of the ACM graphics classes, there is also a hierarchy with the UI interactor classes. The most important ones are summarized in the following diagram, but there are many more:



.

Layout



We already learned about the different regions SOUTH, NORTH, and CENTER. Those are a special feature of the BorderLayout. There are several other layouts besides the BorderLayout. Layouts are about how to "layout" several widgets on the screen. The following are some of the more important layouts:

- **BorderLayout:** here there are three regions and widgets must be explicitly assigned to a region.
- **FlowLayout:** is the simplest layout, widgets are simply laid out side by side from left to right.
- **GridLayout:** the available space is divided into equally separated rows and columns, e.g. 3 by 2.

Let's look at a few examples. To use the BorderLayout we would use the following code:

```
setLayout('border');
addWidget(new JSButton("0"), 'NORTH');
addWidget(new JSButton("1"), 'SOUTH');
...
```

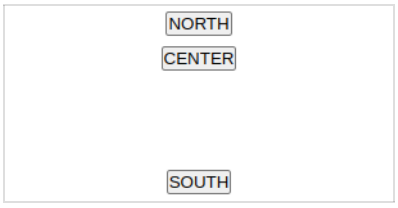
For the FlowLayout the code looks like this:

```
setLayout('flow');
addWidget(new JSButton("0"));
addWidget(new JSButton("1"));
...
```

and the GridLayout is used in this fashion:

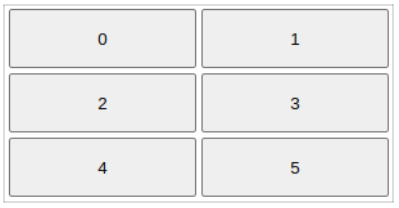
```
setLayout('grid', 2);
addWidget(new JSButton("0"));
addWidget(new JSButton("1"));
...
```

The following depicts how the different layouts look graphically:



Try it

Try it



Try it

.

JSPanel

What the GCompound was for graphics programs is the JSPanel for UI programs: it allows us to combine several widgets into a new widget. Details about the JSPanel class can be found below in the project "Quiz".

.

REVIEW

Congratulations! As we will soon see, we now have the tools to create almost any graphical user interface (UI). We know how to work with

- Labels,
- Buttons,
- Textfields,
- Checkboxes,
- Radiobuttons,
- and Comboboxes.

We also learned about different layouts and briefly got to know the JSPanel.

Equally important, however, was the deepening of our understanding of global, instance and local variables.

.

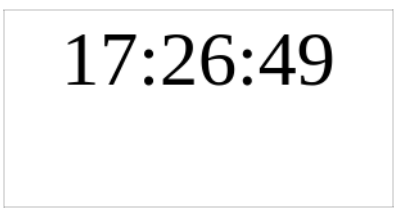
PROJECTS

In the projects of this chapter we will create simple UIs. Some we will need later again. There's a lot to do, so let's get started.

.

Clock

We want to write a little digital clock. For this we use the Date class and its methods getHours(), getMinutes() and getSeconds(). For the display itself we use a JSLabel. And of course the text of the JSLabel should change once per second (better twice). For this you can use the setText() method of the JSLabel.



It may make sense to write a method padWithZeros() that ensures that "06" minutes are displayed instead of "6" minutes.

Try it

.

.

WordGuess

Try it

WordGuess is a graphical version of Hangman from the last chapter. It's about guessing a word by typing letters.

We start by selecting a random word with the pickRandomWord() method. This word should be saved in the instance variable wordToGuess. Then we should create an instance variable, wordShown, with as many dashes, '-', as there are characters in the wordToGuess. Next, we create a JSLabel and add it to the northern region:

```
wordLbl = new JSLabel(wordShown);
wordLbl.addStyle('font: 60px Courier;');
addWidget(wordLbl, 'NORTH');
```

What remains to be implemented is the method keyTyped(KeyEvent e). It is called when the user presses a key. Similar to Hangman, if a key was pressed, we check if the letter is in wordToGuess, and update the label if it is. Of course it also makes sense to count how many attempts were needed to guess the word correctly.

.

StopWatch

Try it

A stopwatch needs a higher accuracy than what the Date class provides. For this purpose we can use the now() method of the Date class,

```
startTime = Date.now();
```

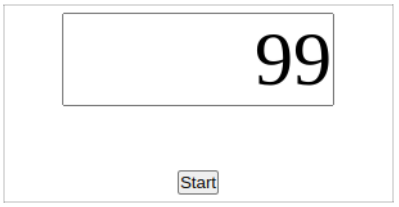
which returns the milliseconds that have elapsed since 0 o'clock on January 1, 1970, UTC.

We can use this to measure how much time has passed. We simply remember the time, when the user presses the "Start" button, and subtract from it the current time. To get the time in seconds, we simply divide this by 1000. If, however, we take this modulo 1000 we get only milliseconds.

The display should be animated, so it makes sense to put the whole thing in a game loop. It also makes sense to have a delay of 20ms, otherwise the refresh of the display is to slow, and we will see nothing. And we should include two buttons, one to start and one to pause.

.

CountDown



Try it

The CountDown works similar to the stopwatch. Instead of a JSLabel we use a JSTextField. The advantage is that you can edit it, i.e. you can set the number from which you want to count backwards. As soon as the user presses the "Start" button, the countdown should begin.

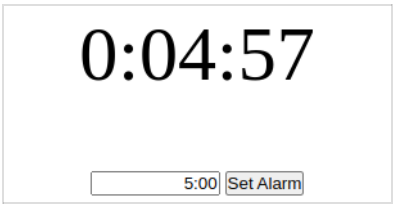
.

.

.

.

AlarmClock



Try it

Writing an alarm clock that uses the hour:minute:second format is surprisingly complicated. However, if you already have the methods for the conversion from hours:minutes:seconds to seconds convertTimeInSeconds() and from seconds to hours:minutes:seconds, convertSecondsInTime(), then it is not that difficult, and actually similar to the CountDown project.

For this program we use a large JSLabel, which we place in the north. There is also a JSTextField in the south for entering the alarm time in the hours:minutes:seconds format. And there is a JSButton to start the alarm.

It makes sense to use two instance variables:

```
let alarmTime = -1;
let alarmStarted = false;
```

The first is simply the time in seconds, and the second is used to tell the game loop to do something:

```
function draw() {
    if (alarmStarted) {
        ...
    }
}
```

If the JSButton is pressed, then alarmTime is set to the seconds, and alarmStarted is set to true, so that the game loop knows that it should now display something.

.

Editor

Try it

Next on our list of things to do is a text editor. This consists of a JSTextField for the file name and two JSButtons, one for loading files, the other for saving files. We place these three widgets in the south, the lower part.

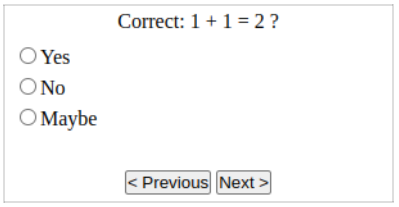
In the middle part, the CENTER region, we place a JSTextArea:

```
display = new JSTextArea("Enter text here...",10,20);
display.addStyle('font: 18px Courier;');
display.addStyle('width: 99%');
display.addStyle('height: 98%');
addWidget(display, 'CENTER');
```

A JSTextArea is like a JSTextField, only that you can enter multi-line text. In the next chapter we will learn how to load and save files. The 99% and 98% are needed, because I do not really understand CSS...

.

Quiz



Try it

We want to write the UI for a MultipleChoice quiz. This consists of a question, i.e. a JSLabel, which we place in the upper area (NORTH). Then below follow the possible answers. These are, of course, radio buttons. Since they belong together, we group them into a button group. The radio buttons are placed in the middle area (CENTER). Finally, we also want to add two navigation buttons at the bottom, the SOUTH. The program should have no further functionality, this comes in the chapter after the next one.

This program is a nice example how to use JSPanels. Multiple-choice questions do not always consist of three answers. Sometimes they are less, sometimes more. So it makes sense to summarize the questions and insert them into a JSPanel,

```
let answersPnl = new JSPanel();
answersPnl.setLayout('grid', 1);
...
let btn1 = new JSRadioButton(answer1);
btn1.addStyle('justify-self: start;');
answersPnl.add(btn1);
...
addWidget(answersPnl, 'CENTER');
```

and then insert the JSPanel into the center region.

.

DrawingEditor

In the last chapter of this book we want to write a drawing editor. Until then, we can do some preparatory work. The idea is that we can choose between the shapes rectangle and circle with two radio buttons. Additionally we want to be able to set whether these shapes should be filled or not. This is best done with a check box. And finally, we want to be able to determine the color of the shapes with a combo box.

Try it

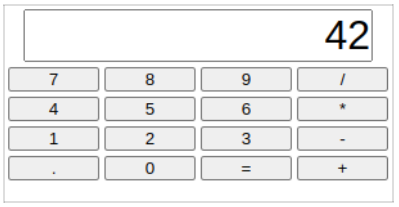
.

CHALLENGES

.

Calculator

Our next project is a small calculator. This consists of a JSTextField (display) and 16 JSButtons. Best is to place the JSTextField in the north, and the JSButtons in a 4x4 GridLayout.



Try it

When programming the logic of this calculator, we need to conisder the following: First, it makes sense to introduce two instance variables:

```
let operand1 = 0;
let operation = '+';
```

For example, if we calculate "6 - 2" with our calculator, we first enter the number "6", then the "-" and then the "2". So we have to remember both the "6" and the "-" in between, hence the two instance variables.

In the actionPerformed() method we must distinguish between the "=" character, the operators ("+", "-", "*" and "/"), and digits.

When digits are entered, we simply append them to the display:

```
let cmd = e.getActionCommand().charAt(0);
display.setText(display.getText() + cmd);
```

If an operator was entered, then we have to set the instance variables operand1 and operation, so

```
operand1 = Number(display.getText());
```

```
display.setText("");
operation = cmd;
```

And if the "=" sign was pressed, then we have to perform the calculation and display the result:

```
let operand2 = Number(display.getText());
let result = calculate(operand1, operand2, operation);
display.setText("" + result);
```

All that is left to do, is to implement the method calculate(double operand1, double operand2, char operation).

.

QUESTIONS

1. There are local variables, instance variables and constants. Explain the difference.

2. In the following example there are several variables, some with the same name. Describe how the variables are related and which are valid where.

```
class Lifetime {

  function run() {
    let i = 3;
    cow(i);
  }

  function cow( n ) {
    for (let i=0; i<3; i++) {
        ...
    }
  }
}
```

3. Name three different LayoutManagers.

4. Sketch what the UI for the following code would look like.

```
...
face = new JSLabel("0:00:00");
face.addStyle('font: 60px SansSerif;');
addWidget(face, 'NORTH');

tfAlarm = new JSTextField(10);
tfAlarm.addStyle('text-align: right;');
addWidget(tfAlarm, 'SOUTH');

btnStart = new JSButton("Set Alarm");
addWidget(btnStart, 'SOUTH');
...
```

5. If you click on a JButton, what kind of event is triggered?

.

.

.

.

REFERENCES

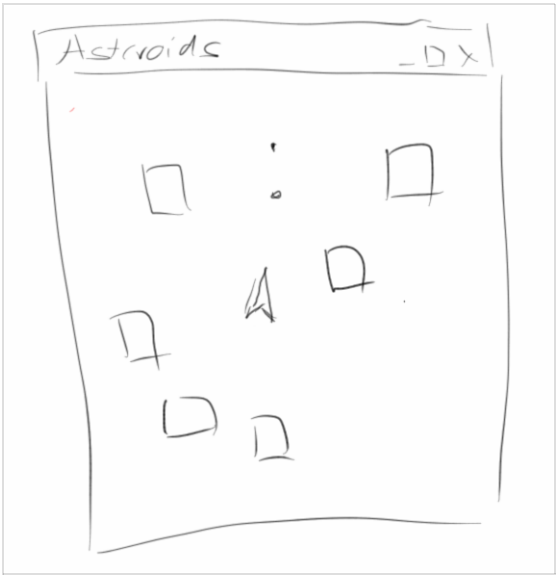
In this chapter the preferred reference is the book by Eric Roberts [1]. A nice but challenging tutorial is that by Oracle, the creators of Java [2].

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

[2] The Swing Tutorial, docs.oracle.com/javase/tutorial/uiswing/

.

.



ASTEROIDS

This chapter is again about graphics and games. But before we get started we first need to learn about arrays. This then allows us to manipulate images, which are nothing but two-dimensional arrays. Next we spend some more time on object orientation, namely inheritance and composition. And we will also learn how to work with key events.

.

Arrays

What are arrays? An egg carton is an array. It's an array for ten eggs. That does not necessarily mean that there are always ten eggs in there, sometimes there are only three eggs in it.

Obviously, arrays are quite practical and therefore we will take a closer look at arrays in JavaScript. Suppose we wanted to create an empty array, then we would write:

```
let eggs = [];
```

The square brackets say that we declared an empty array. We can also create an array that has stuff in it, like numbers:

```
let eggs = [1, 2, 3, 4];
```

In this case, the array has four numbers in it. We can create arrays of any data types, e.g. we could also create an array with four GOvals:

```
let circles = [];
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);
```

Arrays have the important property that they are ordered, i.e. they are numbered consecutively starting with 0.

.

Working with Arrays

After we have declared and created an array, we have to fill it with values. We can do this by hand,

```
let eggs = [];
eggs[0] = 0;
eggs[1] = 2;
eggs[2] = 4;
...
eggs[9] = 18;
```

Meaning we assign the first element in the array (element number 0) the value 0, the second element the value 2 etc. We can also do the assignment with a loop:

```
let eggs = [];
for (let i=0; i<5; i++) {
    eggs[i] = await readInt("?");
}
```

or as we have seen above:

```
let eggs = [ 2, 4, 6, 8 ];
```

If we want to access an element, we have to enter its house number. So we access the third element with:

```
println( eggs[2] );
```

If we want to output all elements, it is best done with a loop:

```
for (let i=0; i<eggs.length; i++) {
            println( eggs[i] );
}
```

.

Exercise: MonthName

A useful example is the conversion of the month as a number, e.g. 12, into the months name, e.g. December. You could do this with a lengthy if or switch condition, but you can also do it very elegantly with arrays:

```
Enter number of month (1=January):
12
December
```

Try it

```
const monthName = ["January", "February", "March", "April",
    "May", "June", "July", "August", "September", "October",
    "November", "December"];

async function setup() {
    createConsole();

    let monthNr = await readInt("Enter number of month (1=January): ");
    println(monthName[monthNr - 1]);
}
```

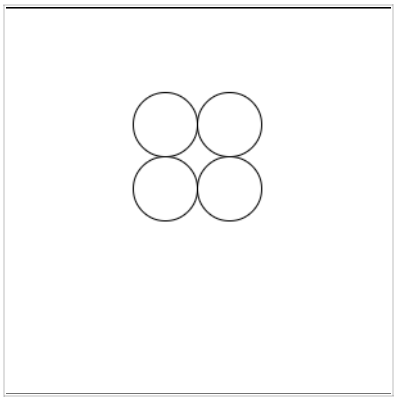
.

.

.

Exercise: Array of Objects

Arrays are not only useful with numbers and strings, but also with objects. Let's say we want to create an array with four ovals in it, The following code would do that:



Try it

```
function setup() {
    createCanvas(300, 300);
    frameRate(5);

    let circles = [];
    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);
    add(circles[0]);
    add(circles[1]);
    add(circles[2]);
    add(circles[3]);
}

function draw() {
    update();
}
```

.

Multidimensional Arrays

One-dimensional arrays are fun and save us a lot of paperwork. But two-dimensional arrays are even cooler. We start quite simple with the game of chess, which can be represented by an 8 by 8 array of chars:

```
rnbqkbnr
pppppppp



PPPPPPPP
RNBQKBNR
```

Try it

```
const 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']];
```

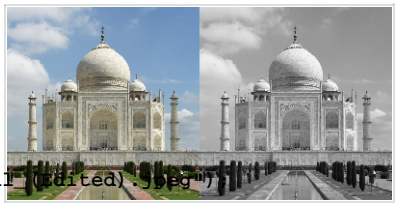
Lowercase letters stand for black, and uppercase letters for white. If we want to display the playing field, then we could do that with two nested for loops:

```
function printChessBoard() {
    for (let i = 0; i < 8; i++) {
        for (let j = 0; j < 8; j++) {
            print(chess[i][j]);
        }
        println();
    }
}
```

.

.

Exercise: GrayImage



Try it

Images are just two-dimensional arrays. As a little exercise we want to convert a color image into a gray image. First we load the image using the GImage class:

```
let image = new GImage("Ch7_Asteroids/Taj_Mahal_(Edited).jpg");
```

Next, we need to get to the pixels. This can be done with the method *getPixelArray()* of the class GImage:

```
let pixels = await image.getPixelArray();
let width = image.width;
let height = image.height;
```

This gives us a two-dimensional array of numbers. Notice the *await*, which is neccessary because sometimes it takes a little longer for an image to load, and we should not access the pixels of an image before it is loaded.

One pixel consists of four numbers: the first for the red color, second for the green color, third blue color and the fourth is the alpha value of a pixel, that is its transparency. Let's say we want to access a pixel at position x=5 and y=22, then the following formula gives us the position of that pixel:

```
let i = (y * width + x) * 4;
```

To get to the color values, we use this index:

```
let red = pixels[i + 0];
let green = pixels[i + 1];
let blue = pixels[i + 2];
let alpha = pixels[i + 3];
```

This is how we read the pixels. But we want to create a new gray image. Hence, first we need to create a new empty pixel array:

```
let pixelsGray = [];
```

Then we use the formula that Gimp uses to calculate the gray value of a pixel [2]:

```
let lum = Math.trunc(0.21 * red + 0.72 * green + 0.07 * blue);
```

and then we put these values into the new pixel array:

```
pixelsGray[i + 0] = lum;
pixelsGray[i + 1] = lum;
pixelsGray[i + 2] = lum;
pixelsGray[i + 3] = alpha;
```

This we do for all the pixels in the array using a nested for-loop:

```
for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
                ...
        }
}
```

Finally, we turn this array back into a new GImage:

```
let gray = new GImage(width, height);
gray.setPixelArray(pixelsGray);
add(gray, 200, 0);
```

.

Object Orientation

In the second part of this chapter we want to deepen our understanding with regards to object orientation. The two big themes are inheritance ("is a" relationship) and composition ("has a" relationship). We start with a little game, the MarsLander.

.

Exercise: MarsLander

Try it

In the next decade, Elon Musk wants to send the first humans to Mars. Karel volunteered and needs to practice landing. For this purpose, we need to write a MarsLander simulator. It's about landing a spaceship safely on Mars. The plan is to use the arrow keys (up and down) to slow down or accelerate our spaceship. If the touchdown speed is to high, Karel dies.

We will use the top-down approach, starting with the draw() method:

```
function draw() {
    if (spaceShip != null) {
        moveSpaceShip();
        checkForCollision();
    } else {
        displayGameOver();
        noLoop();
    }
    update();
}
```

As usual, there is also a setup() that creates the spaceship. The game loop looks exactly like our last animation, billiards. What's interesting now is that we no longer have an infinite loop, but a loop with the abort criterion: namely if there is no more SpaceShip, i.e. spaceShip == null, then the game should stop. The *noLoop()* statement stops the looping.

We have three instance variables,

```
let spaceShip;
let vy = 0;
let vx = 0;
```

i.e. the spaceShip, together with its velocities vx and vy.

In the setup() the spaceShip is initialized. The code is identical to the one from the exercise in chapter two, nothing new here,

```
function setup() {
    createCanvas(APP_WIDTH, APP_HEIGHT);
    frameRate(5);

    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, (APP_WIDTH - SPACE_SHIP_SIZE) / 2, SPACE_SHIP_SIZE);
}
```

The moveSpaceShip() method is absolutely trivial:

```
function moveSpaceShip() {
    vy += GRAVITY;
    spaceShip.move(vx, vy);
}
```

The speed of our space ship increases with every step by the GRAVITY of Mars, and it is moved by its velocities.

In the checkForCollision() method we check if we have already reached the surface of Mars:

```
function checkForCollision() {
    let y = spaceShip.getY();
    if (y > (APP_HEIGHT - SPACE_SHIP_SIZE)) {
        spaceShip = null;
    }
}
```

If so, we simply set the spaceShip to null. Here "null" means: "not initialized" or "there is none" or "does not exist". We can set objects explicitly to null, which means that we delete the object. In our example, we use this to end our game loop.

What remains are the key events: similar to the mousePressed() method, there is a keyPressed() method:

```
function keyPressed() {
    switch (keyCode) {
        case 38: // up
            vy--;
            break;
        case 40: // down
            vy++;
            break;
    }
}
```

Of course we want to know which key was pressed and this we get from the keyCode. Each key has its own key code, and for the up key this is 38 and for the down key this is 40.

Now we can play or rather train for our Mars mission.

.

Inheritance

What does MarsLander have to do with inheritance? Not much yet. But let's look at the code. What should bother us a little bit are the following three lines,

```
let spaceShip;
let vy = 0;
let vx = 0;
```

because vx and vy are the speed of the spaceShip, meaning they actually belong to the spaceShip. Suppose we had several spaceShips, or we had a lot of asteroids moving around, then we

would have a lot of vx's and vy's. And it's going to be very confusing and ugly. Try it

To prevent this we do the following: we declare a new class called GSpaceShip and we put everything that has to do with the spaceShip into this class:

```
class GSpaceShip extends GPolygon {

    constructor(x, y) {
        super(x, y);
        super.addVertex(0, -SPACE_SHIP_SIZE);
        super.addVertex(-2 * SPACE_SHIP_SIZE / 3, SPACE_SHIP_SIZE);
        super.addVertex(0, SPACE_SHIP_SIZE / 2);
        super.addVertex(2 * SPACE_SHIP_SIZE / 3, SPACE_SHIP_SIZE);

        this.vy = 0;
        this.vx = 0;
    }

    move() {
        this.vy += GRAVITY;
        super.move(this.vx, this.vy);
    }
}
```

First, we see that GSpaceShip is a GPolygon, because it says "GSpaceShip extends GPolygon". This means that GSpaceShip inherits all properties and methods from GPolygon. That is why inheritance is also said to be an "is a" relationship.

Second, we see that the global variables vx and vy are now instance variables of the spaceShip, so they are where they belong.

Third, we take a look at the constructor: there we see a "super()" in the first line. The method super() does nothing else but call the constructor of the superclass, the parent class. In our case, the GPolygon(). Then we see how we add vertices to ourselves (we are now a GPolygon). So in the constructor we create our spaceship appearance.

Finally, we see that we have added a new method called move(). Since GSpaceShip now knows its own speed, it can also move itself.

Inheritance therefore has many advantages: above all, it leads to classes becoming more independent and having fewer dependencies. It is also said that the class takes responsibility for its own attributes (variables) and behavior (methods). These lesser dependencies also lead to a smaller coupling, which makes our code less complicated.

Let's have a look at the simplifications in MarsLander2. First, we only need one instance variable:

```
let spaceShip;
```

and also the setup() and moveSpaceShip() methods become much cleaner:

```
function setup() {
    createCanvas(APP_WIDTH, APP_HEIGHT);
    frameRate(5);

    spaceShip = new GSpaceShip();
    add(spaceShip, (APP_WIDTH - SPACE_SHIP_SIZE) / 2, SPACE_SHIP_SIZE);
}

function moveSpaceShip() {
    spaceShip.move();
}
```

This is pretty cool. We will really come to appreciate these simplifications when it comes to programming the

Asteroids game.

.

Composition

Try it

The second important concept of object orientation is composition. As we have seen, you can create new classes (GSpaceShip) by inheritance from an existing class (GPolygon). But we can also create new classes by composing them from several existing classes.

As an example we write a class GSmiley.

```
class GSmiley extends GCompound {

    constructor(SIZE) {
        super();

        let face = new GOval(SIZE, SIZE);
        face.setFilled(true);
        face.setFillColor(Color.YELLOW);
        this.add(face);

        let leftEye = new GOval(SIZE / 10, SIZE / 10);
        leftEye.setColor(Color.GREEN);
        this.add(leftEye, SIZE / 4, SIZE / 4);
        let rightEye = new GOval(SIZE / 10, SIZE / 10);
        rightEye.setColor(Color.RED);
        this.add(rightEye, 3 * SIZE / 4, SIZE / 4);

        let mouth = new GArc(SIZE / 2, SIZE / 2, 225, 90);
        this.add(mouth, 0.3 * SIZE, 0.3 * SIZE);
    }
}
```

Our GSmiley consists of different components, so it has a face, a leftEye, a rightEye and a mouth. In the constructor we create a new object from several old objects. That's composition. That's also why we say that composition is a "has a" relationship, because GSmiley has a face, a leftEye, a rightEye and a mouth.

.

GCompound

We can also mix inheritance and composition. If we add "extends GCompound" to the class declaration for the GSmiley example, then we can also use GSmiley in our MarsLander. If we simply replace "GPolygon" with "GSmiley" in our first version of the MarsLander, the program works like before, only our spaceship now looks like a smiley.

.

Inheritance vs Composition

When should we use inheritance and when composition? A rule of thumb is, if possible, use composition. This has to do with the fact that there is no multiple inheritance in JavaScript. So a class can't have two parents. This restriction does not apply to composition, in principle a class can consist of any number of components.

.

.

.

A small note: by multiple inheritance we mean that a class has several parent classes. That's not allowed. But it is quite possible that a class has a parent class, and this parent class again has a parent class, so to speak the grandparent class of the original. For example, GObject is the grandparent class of the GSpaceShip class. That's allowed.

.

.

Review

With the principles of inheritance and composition we have reached and cracked the core of object orientation. We have learned how to give an existing class additional properties through inheritance. A GPolygon cannot move independently because it has no speed. The class GSpaceShip, which is actually also a GPolygon, knows its own speed and can move by itself. We have also seen that we can assemble a new class from several existing classes via composition. Both are very useful, as we will see.

In addition, we have learned a few other useful things, such as

- arrays,
- multidimensional arrays,
- image processing,
- key events,
- and we used the class GCompound.

.

Projects

The projects in this chapter are fun projects. Here we go.

.

Swap

In this project we want to swap two elements in an array. In the array

```
let arr = [0, 2, 4, 6];
```

we want to swap the element at the second position (the "2") with the element at the third position (the "4"). We want to do this with a method *swap(arr)*, which has an array as parameter.

```
0, 4, 2, 6,
```

Try it

Two things we want to learn in this exercise: First, in arrays we always start counting from 0. Second, arrays are passed as reference, i.e. if we pass an array as a parameter to a method, then we pass the original, not a copy. All changes we make to it in the method are permanent, i.e. change the original array.

.

ExamStatistics

As another example for the use of arrays let us store the grades of an exam in an array. Since we do not yet know exactly how many students will take the exam, but it is very unlikely that there will be more than 100, we will create an array for 100 grades:

```
let scores = [];
```

```
Enter grades:
?56
?78
?23
?99
?65
?-1
Number of exams:5
Lowest grade: 23
Highest grade: 99
```

Try it

Then we ask the user to enter the grades. For this the loop-and-a-half is ideally suited. To know when we are done, we agree that entering a "-1" (the sentinel) means that all grades have been entered.

Once we have all the grades in the array, we want to calculate some statistical data about the grades like total number of students who took the exam, the average grade, the lowest grade and the highest grade.

.

PianoConsole

As our first application for arrays we write a small music program. Before being able to play sound, we need to load a library that allows us to do that. Therefore, at the beginning of your code, place the following include() statement:

```
include("./libraries/p5.sound.min.js");
```

```
C4,D4,E4,F4,G4,G4,A4,A4,A4,A4,G4,
```

Try it

(Also, sound may not work on mobile devices, I had problems with an iPad playing any sound.)

Once we have that, we can start by storing our melody in an array,

```
const tune = [0, 1, 2, 3, 4, 4, 5, 5, 5, 5, 4];
```

where 0 refers to C4, 1 to D4, etc. But before playing, we must load the sound files. Those must be preloaded:

```
const songFileNames = "CDEFGAB";
function preload() {
    for (let i = 0; i < songFileNames.length; i++) {
        let soundName = songFileNames.charAt(i) + '4';
        let fileName = 'Pr7_Asteroids/music/' + soundName + '.wav';
        soundFiles[i] = loadSound(fileName);
    }
}
```

Preloading is neccessary, because sometimes the internet is a little slow, and we want to make certain that our program only starts after all the sound files have loaded.

Next we initialize our program. Since console programs usually have no loop, we must turn the loop on, and we also declare and initialize the *currentSong* variable.

```
let currentSong;
function setup() {
    createConsole();
    loop();
    frameRate(5);

    currentSong = soundFiles[0];
}
```

Now we are ready to play our tune:

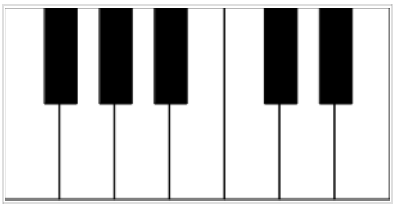
```
function draw() {
    if (!currentSong.isPlaying() && counter < tune.length) {
        let tun = tune[counter];
        print(songFileNames.charAt(tun) + ",");
        currentSong = soundFiles[tun];
        currentSong.play();
        counter++;
    }
```

```
}
```

We wait until a sound file has finished playing, before we move to the next. Our counter helps us to keep track on which sound to play. Obviously, you can play other melodies, add different and more sound files, etc.

.

Piano



Try it

Console applications are always a little more boring, and honestly, who would spend any money on them? But we already wrote a UI for our piano in the second chapter. Of course we want to control our piano with the mouse.

The question that arises is how do we know which key was pressed? Interestingly, we can use the getElementAt() method for this:

```
function mousePressed() {
    let x = mouseX;
    let y = mouseY;
    let obj = getElementAt(x, y);
    if (obj !== undefined) {
        ...
    }
}
```

This gives us the GRect that was pressed. Now there are three alternatives to go on:

1. If the GRect had a name, then we would know which tone to play. We can do this with inheritance: we define a new class GKey, which is a GRect and has an additional attribute for the name.
2. We memorize somewhere the x-coordinate of the keys. With obj.getX() we can get it, and viola we know which key was pressed.
3. Or we keep references to all keys in an array.

Let us take a closer look at this third possibility. For this we need an array as global variable:

```
let keys = [];
```

When we create the keys, we store them in our global array:

```
let keyCounter = 0;
// draw 8 white keys
for (let i = 0; i < 7; i++) {
    keys[keyCounter] = new GRect(WIDTH / 7, HEIGHT - HEIGHT_OFFSET);
    add(keys[keyCounter], i * WIDTH / 7, 0);
    keyCounter++;
}
```

And now we can simply test for equality in our mousePressed() method:

```
function mousePressed() {
    let x = mouseX;
    let y = mouseY;
    let obj = getElementAt(x, y);
    if (obj !== undefined) {
        for (let i = 0; i < songFileNames.length; i++) {
            if (obj == keys[i]) {
                print(songFileNames.charAt(i) + ",");
                currentSong = soundFiles[i];
                currentSong.play();
```

```
                }
            }
        }
    }
```

If we could get this program to work on our mobile phones, we'd be rich! (Next year...)

BTW, only worry about the white keys. The black keys are a little tricky, in a little while we learn even simpler ways to do this.

.

FlippedImage

Try it

When manipulating images we usually use arrays. We have already seen how we can access the pixels of an image. In this example we want to mirror a given image. This can be done horizontally or vertically. To do this, we create a new array for the pixels

```
let pixelsFlipped = [];
```

and use two nested loops

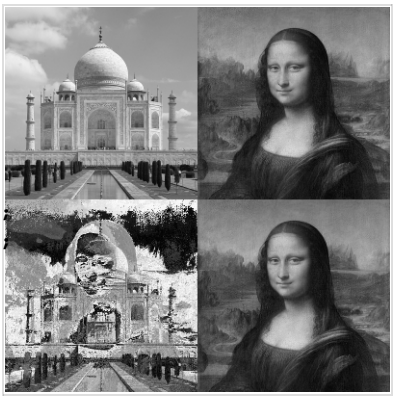
```
for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
        let i = (y * width + x) * 4;
        let j = ((height - y) * width + x) * 4;
        pixelsFlipped[j + 0] = pixels[i + 0];
        pixelsFlipped[j + 1] = pixels[i + 1];
        pixelsFlipped[j + 2] = pixels[i + 2];
        pixelsFlipped[j + 3] = pixels[i + 3];
    }
}
```

to swap the pixels. From the new array we create a new image via

```
let flipped = new GImage(width, height);
flipped.setPixelArray(pixelsFlipped);
return flipped;
```

.

GrayImageXOR



Try it

Steganography is the art of hidden transmission of information [3]. It is interesting that this can be done quite simply with the XOR function, i.e. the Exclusive Or. How do we do that? Let's assume we have two images and their pixel arrays:

```
let pixels1 = await grayImage1.getPixelArray();
let pixels2 = await grayImage2.getPixelArray();
```

then we get the red value for each pixel, for example:

```
let i1 = pixels1[i];
let i2 = pixels2[i];
```

And just as we could add these two values, we could also use the XOR function:

```
let xr = i1 ^ i2;
```

We make a bit-wise XOR of the bits of r1 with those of r2. To test this, we take the Taj and the Mona Lisa as images, and 'XOR' them. The result is a funny mixture, where you can recognize neither one nor the other, or both depending. The interesting thing happens, when we take the pixels of that mixed image and let the XOR function run over it again: then the original image appears again. Interestingly, exactly the opposite. This can also be used for encryption.

The RAID-5 system is based on the same principle and is used to provide failure safety for hard disks [4].

.

ColorImage

Try it

A common use of image manipulation is to reduce the number of colors in an image. This is a nice application for rounding to integers.

```
let r = pixels[i + 0];
r = Math.trunc(r / FACTOR) * FACTOR;
```

Originally r can take values between 0 and 255. If we divide this number by let's say 64, then we only have values between 0 and 3. If we multiply this again by 64, we only have the values 0, 64, 128 and 192. So there are only four red values left.

.

ImageFilterSimple

Try it

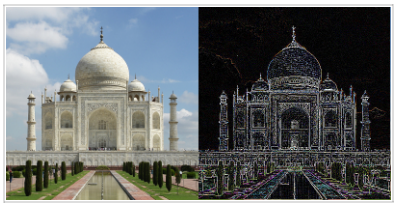
You can also perform many other image manipulations. For example, you can simply subtract adjacent pixels:

```
let n = (y * w + x - 1) * 4;
r[0][1] = pixels[n + 0];
n = (y * w + x) * 4;
r[1][1] = pixels[n + 0];
...
let xx = r[1][1] - r[0][1];
xx *= 10;
...
let nn = (y * w + x) * 4;
pixelsGray[nn + 0] = xx;
```

The result corresponds to a simple edge detection.

.

ImageFilterMatrix



Try it

Much more interesting image manipulations become possible if we realize that arrays are also matrices. You may want to keep this to yourself, or you might get burned [6]. But once you know that, you can do cool things with images. The filters sharpen, blur, edgeEnhance, edgeDetect, or emboss, as they are known from any image processing program, are actually only the application of a convolution matrix on an given image [5]. For example, to sharpen an image, the following matrix will do the trick:

```
let currentFilter = [
        [0, -1, 0],
        [-1, 5, -1],
```

```
            [0, -1, 0]
        ];
        let currentFactor = 1;
```

Executing matrix multiplication (that is, applying the filter to the image) is performed in the following method:

```
        function applyFilterToPixel(x, y, width) {
            let r = 0;
            let g = 0;
            let b = 0;
            for (let i = 0; i <= 2; i++) {
                for (let j = 0; j <= 2; j++) {
                    let n = ((x + i) * width + (y + j)) * 4;
                    r += array[n + 0] * currentFilter[j][i];
                    g += array[n + 1] * currentFilter[j][i];
                    b += array[n + 2] * currentFilter[j][i];
                }
            }

            let nn = (x * width + y) * 4;
            array2[nn + 0] = checkBounds(r / currentFactor);
            array2[nn + 1] = checkBounds(g / currentFactor);
            array2[nn + 2] = checkBounds(b / currentFactor);
            array2[nn + 3] = 255;
        }
```

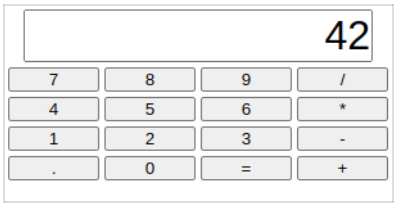
This must be called for each pixel of the original image. Notice, that we made the two pixel arrays global variables:

```
        let array = [];
        let array2 = [];
```

This is actually a pretty advanced program.

.

Calculator



Try it

Applications for arrays are really diverse. Very often they save you from having to write a lot of code. A nice example is the calculator from the last chapter. You can of course create the buttons, i.e. JSButtons, all individually, but you can also make it a bit more effective:

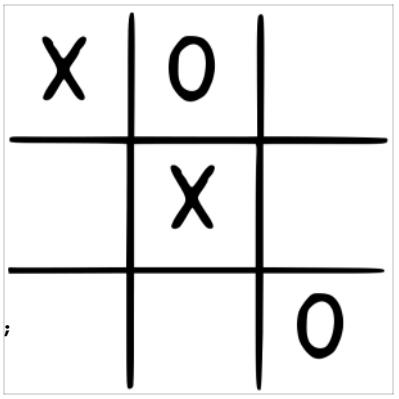
```
        const btnNames = ["7", "8", "9", "/", "4", "5", "6", "*",
            "1", "2", "3", "-", ".", "0", "=", "+"];

        function setup() {
            ...
            let pnl = new JSPanel('grid', 4);
            addWidget(pnl, 'CENTER');
            for (let i = 0; i < btnNames.length; i++) {
                let btn = new JSButton(btnNames[i]);
                pnl.add(btn);
            }
        }
```

.

.

.

TicTacToeLogic



Try it

Arrays can also be useful for games. In the fourth chapter we already wrote the UI for the TicTacToe game. Now we are ready to understand the logic part. The playing field can be understood as a two-dimensional array, which is an instance variable that we initialize in the constructor:

```
class TicTacToeLogic {

    constructor() {
        this.board = [[0,0,0],[0,0,0],[0,0,0]];
    }
        ...
}
```

Originally all values of the playing field are set to 0. If we now mark the squares that player one has occupied with a 1 and the squares that player two has occupied with a 2, then this is a perfect description of the current state of the game.

If we want to test if a certain move is allowed, then we only have to check if the value of the playing field at that position is 0:

```
isMoveAllowed(player, i, j) {
    if (this.board[i][j] == 0) {
        this.board[i][j] = player;
        return true;
    }
    return false;
}
```

If we want to test if a player has won, we have to check if one of the players has occupied a whole vertical, horizontal or diagonal row. For the vertical rows the following will do the trick:

```
checkVerticals() {
    for (let i = 0; i < 3; i++) {
        if ((this.board[i][0] == 1) && (this.board[i][1] == 1)
                                    && (this.board[i][2] == 1)) {
            return true;
        }
    }
    for (let i = 0; i < 3; i++) {
        if ((this.board[i][0] == 2) && (this.board[i][1] == 2)
                                    && (this.board[i][2] == 2)) {
            return true;
        }
    }
    return false;
}
```

We just go through one column after the other (for-loop) and check if all three values are set to 1 (for player 1) or 2 (for player 2).

.

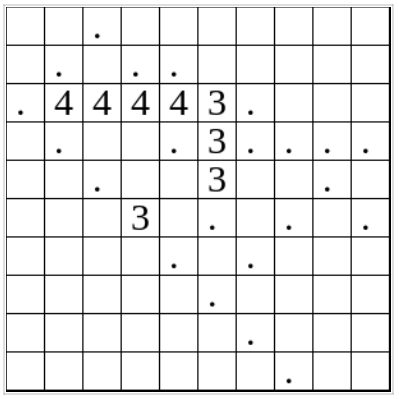
.

.

.

.

BattleShip



Try it

Battleship [7] is another classic game for which arrays are very useful. Our BattleShip game should become a game human versus computer, that means the computer distributes its boats on the playing field and we have to find them.

Just like TicTacToe, we use an array of integers for the playing field:

```
let board = makeArray(BOARD_SIZE, BOARD_SIZE);
```

where we dynamically create the array with the makeArray() function:

```
function makeArray(d1, d2) {
    var arr = [];
    for (let i = 0; i < d2; i++) {
        arr.push(new Array(d1));
    }
    return arr;
}
```

The ships themselves are represented by numbers: 5 stands for an AircraftCarrier, 4 for a Battleship, 3 for a Submarine or a Destroyer and 2 for a PatrolBoat. To determine how many of each type of ship there are, we can use another array:

```
const SHIP_SIZES = [5, 4, 3, 3, 2];
```

That means, if we want to have some more PatrolBoats, just add some more 2's.

In the setup() method,

```
function setup() {
    ...
    drawLines();
    initBoard();
}
```

we draw the playing field and initialize the boats. In the initBoard() method we go through the list of ships (SHIP_SIZES) and add one by one using the placeShip(int shipNr, int shipSize) method. This method can be quite simple, if we place the ships just next to each other. However, then the game becomes boring quickly. On the other hand, we can also place the ships randomly, then things get quite complicated, because the ships are not supposed to 'collide'. For us the simple version is fine.

What remains to be implemented is the mousePressed(). We'll use our integer division trick again:

```
function mousePressed() {
    let i = Math.trunc(mouseX / STEP);
    let j = Math.trunc(mouseY / STEP);
    showLabelAt(i, j);
}
```

Next, we implement the showLabelAt(int i, int j) method: it checks the board array to see if there is a ship at the location the user selected:

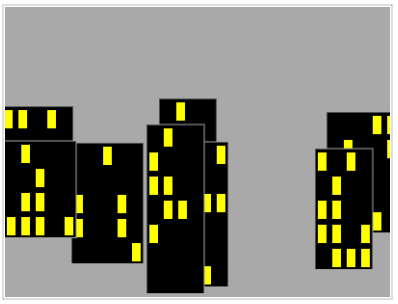
```
function showLabelAt(i, j) {
    let lbl = new GLabel("" + board[i][j]);
    if (board[i][j] === undefined) {
        lbl = new GLabel(".");
```

```
    }
    lbl.setFont("SansSerif-bold-24");
    let x = i * STEP + 7;
    let y = j * STEP + 24;
    add(lbl, x, y);
}
```

This is another trick that can save you lots of unnecessary code. What's left to do is to draw the label at the position where the mouse was clicked. And that's it.

.

CityAtNight



Try it

Reuse is a very central concept of object orientation. This can be achieved with inheritance as well as with composition. Let's start with an example for composition. Let's remember chapter 2, where we programmed a skyscraper. If we want to draw a whole city now, it would be very practical if we could reuse our skyscrapers:

```
const rgen = new RandomGenerator();
function setup() {
    ...
    for (let i = 0; i < 8; i++) {
        let cols = rgen.nextInt(4, 6);
        let rows = rgen.nextInt(4, 8);
        let x = rgen.nextInt(-40, APP_WIDTH - 40); // 120
        let y = rgen.nextInt(APP_HEIGHT / 4, APP_HEIGHT / 2); // 30
        let h = new GSkyscraper(rows, cols);
        add(h, x, y);
    }
}
```

So we would need a class GSkyscraper to draw a skyscraper. Since a skyscraper consists of several GRects, it makes sense, analogous to the GSmiley, to use a GCompound:

```
class GSkyscraper extends GCompound {
        ...
}
```

As with any class, we need a constructor

```
constructor(rows, cols) { {
        ...
}
```

in which we pass the number of window rows and columns we want the skyscraper to have. Depending on whether all skyscrapers should look the same or differently we have to add some randomness to the addWindow() method.

.

SevenSegmentDisplay

Try it

Another nice example of reuse by means of composition is the seven-segment display. Let's remember chapter 2, where we programmed a seven-segment display. If we now need several of these seven-segment displays, e.g. for a counter, a calculator or a clock, then it would be practical if there was a SevenSegmentDisplay class that we could simply use several times, similar to a GRect.

Since a seven-segment display consists of several GRects, it makes sense, similar to the GSmiley, to again use a GCompound:

```
class SevenSegmentDisplay extends GCompound {
        ...
}
```

As with any class, we need a constructor

```
constructor(width, height, ledWidth) {
        ...
}
```

in which we ideally specify the width and height of the display, as well as the width of the LEDs. The constructor should then construct the display from GRects.

Really practical would be a displayNumber(char c) method,

```
displayNumber(c) {
    c = '' + c;
    this.turnAllSegmentsOff();
    switch (c) {
        case '0':
            const code0 = [1, 1, 1, 1, 1, 0, 1];
            this.turnSegmentsOn(code0);
            break;
        case '1':
            ...
    }
}
```

to which you simply pass a number, and which then displays it. The turnSegmentsOn() method could look like this:

```
turnSegmentsOn(code) {
    if (code[0] == 1) {
        this.upperFrontVertical.setColor(this.colorOn);
    }
    ...
}
```

The SevenSegmentDisplay can then be easily used in a graphics program:

```
function setup() {
    ...
    ssd1 = new SevenSegmentDisplay(40, 80, LED_WIDTH);
    add(ssd1, X_OFFSET + LED_WIDTH, LED_WIDTH);
    ssd2 = new SevenSegmentDisplay(40, 80, LED_WIDTH);
    add(ssd2);
    ssd2.move(X_OFFSET + 40 + 3 * LED_WIDTH, LED_WIDTH);
}
```

Extension: Instead of the JSTextField you could also use the SevenSegmentDisplay for the Calculator.

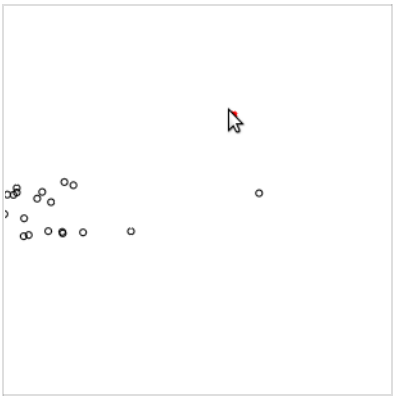
.

.

.

.

BirdFlocking



Try it

Swarming behavior can be observed in fish, birds and many other animals. Interestingly, swarm behavior can be simulated relatively easily. The individuals in a swarm, sometimes called boids, only have to follow three simple rules [11]:

- Separation: keep some distance from your neighbors if you get too close (short range repulsion).
- Alignment: move roughly in the direction of your neighbors.
- Cohesion: move roughly towards the common center of your neighbors (long range attraction).

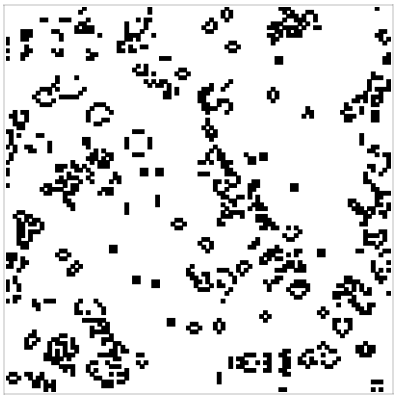
The simulation is similar to the Planets project (later in this chapter), with the subtle difference that instead of Newton's gravity, the boid rules apply.

.

.

.

GameOfLife



Try it

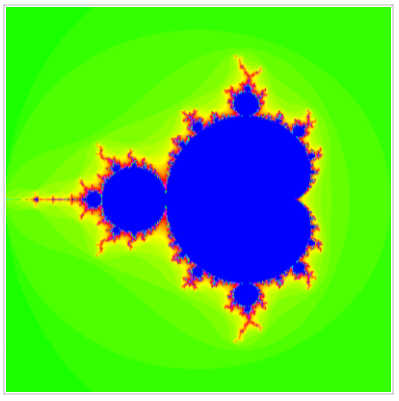
The greatest genius of the last century, John von Neumann, tried to construct a hypothetical machine that could make copies of itself. He succeeded in doing so, but the mathematical model of his machine had very complicated rules. The British mathematician John Horton Conway succeeded in drastically simplifying Neumann's ideas in the early 1970s, now known as Conway's Game of Life [8].

The universe of the Game of Life is a two-dimensional grid of square cells (GRects), each of which can be in one of two possible states: alive (black) or dead (white). Each cell has eight neighbors, and depending on the state of the neighbors, the state in the next round is decided according to the following rules:

- every living cell with less than two living neighbors dies (sub-population),
- every living cell with two or three living neighbors lives,
- every living cell with more than three living neighbors dies (overpopulation), and
- every dead cell with exactly three living neighbors becomes a living cell (reproduction).

.

Mandelbrot



Try it

The Mandelbrot set is named after the French mathematician Benoît Mandelbrot. This set is a so-called fractal, but most people just find it pretty [9]. The mathematical equation behind the Mandelbrot set is very simple:

```
z_n+1 = z_n * z_n + c
```

where z and c are complex numbers. This equation is an iteration, i.e. if we know z_n, then we can calculate z_n+1. The initial conditions are that z_0 should be zero and c is the point in the complex plane for which the color is to be calculated. So if we think in x- and y-coordinates, then

```
c = x + i y
```

is the initial condition. All that is needed is the abort criterion, when should we stop the iteration? Either if z*z >= 4 or if the number of iterations is greater than a maximum value:

```
while (x * x + y * y < 4 && iteration < MAX_ITERATION) {
    ...
    iteration++;
}
```

To make the whole thing look nice, we take the number of iterations and encode them in color:

```
let color = RAINBOW_COLORS[iteration % RAINBOW_NR_OF_COLORS];
```

RAINBOW_COLORS is a color array that we can initialize at will. Last but not least we need a setPixel() method, which does not exist in the ACM Graphics library. We simply draw little GRects:

```
function setPixel(x, y, color) {
    let i = (int)(((x - xMin) * WIDTH) / (xMax - xMin));
    let j = (int)(((y - yMin) * HEIGHT) / (yMax - yMin));
    let r = new GRect(1, 1);
    r.setColor(color);
    add(r, i, j);
}
```

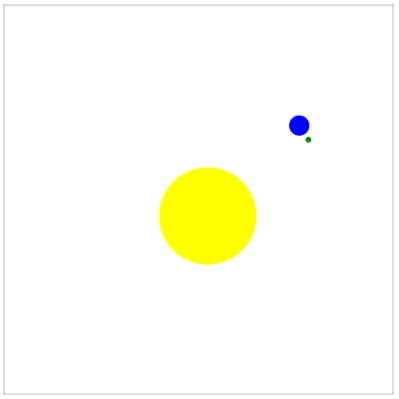
This is not the fastest and most effective way, but it works.

.

CHALLENGES

.

Planets



Try it

A nice example for reuse by inheritance is a small simulation of the sun-earth-moon system. Visually, planets are nothing more than GOvals. But planets move, i.e. they have a speed. GOvals have no velocity. So we need a GOval with velocity. This is exactly what inheritance can do for us:

```
class GPlanet extends GOval {
    constructor(size) {
        super(size, size);
        this.vx = 0;
        this.vy = 0;
    }

    move() {
        super.move(this.vx, this.vy);
    }
}
```

GPlanet is a GOval, but has an additional velocity, i.e., vx and vy. In the constructor we simply call the constructor of the superclass, the constructor of GOval, which creates a GOval with a given height and width. Otherwise we just need a move() method to move our planet.

In our Planets GraphicsProgram we create three planets in the setup(), so

```
function 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 moon
    ...
}
```

Here we set the radius of the sun equal to the mass of the sun. That's not entirely true, but not too bad for the simulation. Next we look at the game loop:

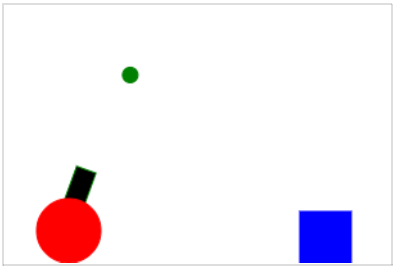
```
function draw() {
    sun.move();
    earth.move();
    moon.move();
    calculateNewVelocities(sun, earth);
    calculateNewVelocities(sun, moon);
    calculateNewVelocities(earth, moon);
    update();
}
```

As usual in the game loop, we first move the individual planets and then calculate the new velocities. The calculateNewVelocities() method looks a bit complicated, but is nothing else than Newton's law of gravity.

This example shows very nicely that simulations are a little tricky: after the second orbit around the sun, our moon leaves us forever... hasta la vista.

.

AngryCanon



Try it

Our next project was inspired by a popular game with birds and pigs. As usual, we need to simplify things a little. The target is a blue GRect, which we should hit with a bullet (green GOval). The bullet is shot by a cannon.

The most difficult thing about this game is the cannon: because we want to be able to change its direction, we need to be able to turn it. However, only the GPolygon offers this functionality, meaning it has a rotate() method. Thus our cannon needs to be a GPolygon. In order for this to look a little more pleasing, we hide part of the cannon behind a red GOval.

In the setup() method we create the cannon and the target.

Next we write the keyPressed() method: here we want to rotate the cannon either left or right depending on the KeyCode,

```
function keyPressed() {
    switch (keyCode) {
        case LEFT_ARROW:
            angle += 5;
            canon.rotate(5);
            break;
        case RIGHT_ARROW:
            angle -= 5;
```

```
                canon.rotate(-5);
                break;
            case 32:
                fireBullet();
                break;
        }
    }
```

If the player presses the space bar we want to fire the bullet. We need the angle angle as an instance variable so that we can set the initial velocity of the bullet when firing:

```
    function fireBullet() {
        if (bullet == null) {
            vx = -Math.sin(angle * Math.PI / 180) * BULLET_SPEED;
            vy = -Math.cos(angle * Math.PI / 180) * BULLET_SPEED;

            bullet = new GOval(BULLET_SIZE, BULLET_SIZE);
            bullet.setColor(Color.GREEN);
            bullet.setFilled(true);
            add(bullet, 50 - BULLET_SIZE / 2, APP_HEIGHT - 30 - BULLET_SIZE);
            sendToBack(bullet);
        }
    }
```

Notice the sendToBack() function, that we call after adding the bullet to the canvas. It changes the z-order of the objects, and makes sure that the collision detection works properly.

The movement of the bullet itself is then calculated in the game loop:

```
    function draw() {
        if (bullet != null) {
            moveBullet();
            collisionWithWalls();
            collisionWithTarget();
        }
        update();
    }
```

In moveBullet() we move the bullet and let gravity do its work:

```
    function moveBullet() {
        bullet.move(vx, vy);
        vy += GRAVITY;
    }
```

If there are collisions with the walls (i.e. up, right, left or down), the bullet simply disappears:

```
    removeObj(bullet);
    bullet = null;
```

If there are collisions with the blue target, the bullet and the target disappear, and the game is over.

.

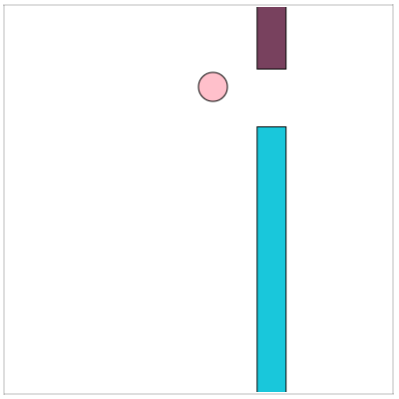
.

.

.

.

FlappyBall



Try it

This project is inspired by a game with a bird. However, our bird is just a GOval. We can control the bird with the keyboard, more precisely the space bar. And the bird has to fly through an obstacle, two moving GRects.

For the game we need a bird (ball) and a two-part wall as instance variables:

```
let ball;
let upperWall;
let lowerWall;
```

In setup() we initialize them: The ball is placed in the middle, the two rectangles to the right edge. The position of the gap should be random, but the width of the gap should be twice the diameter of the ball.

Next follows the game loop:

```
if (alive) {
    moveBall();
    moveWall();
    checkForCollision();
    update();
}
```

The ball only moves up or down. Normally gravity acts on it, so it usually falls down. The wall moves with constant speed from right to left. When it reaches the left edge of the screen, it simply disappears and a new wall appears on the right edge.

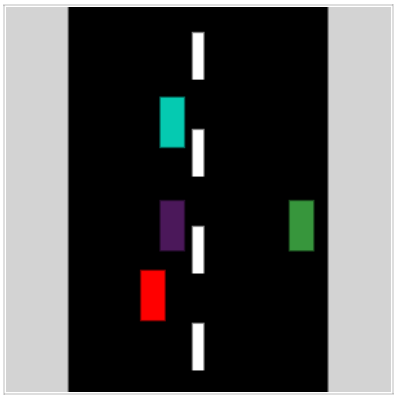
As far as collisions are concerned, we have to check for collisions with the walls: if there is one, the game is over. In case of collisions with the ground, it makes sense to simply set the ball speed to 0 and position the ball at the bottom of the screen.

Still thinking about what to do when the space bar is pressed? This is surprisingly simple:

```
function keyPressed() {
    ballVel = -3.0;
}
```

.

SpeedRace



Try it

The first version of Speed Race appeared in 1974 written by Tomohiro Nishikado, the author of Space Invaders [10]. In a nutshell, it is GTA 0.1, a car racing game of the first generation. It's interesting how easy it is to outsmart the brain: just a few white rectangles that move from top to bottom, and you think you're driving on a road!

The game actually consists of a whole lot of rectangles. The first is the road: it consists of two parts: a large black rectangle that does nothing and the middle lane that consists of several (five) white rectangles that move at a constant speed from top to bottom.

This is followed by your own car: a red rectangle that can only be steered to the left and right. And finally the other cars (otherCars): they are only colored rectangles, which also move from top to bottom through the screen. In setup() all the rectangles are created, notice that the order they are added makes a difference in appearance.

The code for the game loop is also limited:

```
function draw() {
    moveRoad();
    moveCars();
    checkForCollisionCarsWithWall();
    update();
}
```

First we move the road. This is actually totally trivial if we remember our friend the remainder operator:

```
function moveRoad() {
    for (let i = 0; i < NR_OF_LANES; i++) {
        middleLane[i].move(0, CAR_SPEED);
        let x = middleLane[i].getX();
        let y = middleLane[i].getY() + LANE_LENGTH;
        middleLane[i].setLocation(x, y % SIZE - LANE_LENGTH);
    }
}
```

No need for an if-else. Even easier is moveCars(), we simply move one car after the other. In checkForCollisionCarsWithWall() we just want to find out if one of the otherCars has left the screen below: then we simply send it to the top again, but at a different, random x-position.

Remains the keyPressed() method: if the player presses the left arrow key (keyCode = 37), then we simply move the car to the left by 5 pixels, if he presses the right arrow key (keyCode = 39), we move it to the right by 5 pixels. Could have been more complicated.

.

GeometryRun

Try it

Most of us have suffered permanent brain damage from math lessons at school, so in general we avoid geometric objects like the plague. The point of this game is that we (a green GOval) must avoid colliding (using the space bar) with the incoming geometric objects (GRects) at all cost.

As usual, we first consider which instance variables are necessary:

```
let obstacles = [];
let runner;
```

The Geometry class is simply a GOval, the GeometryObstacle is a GRect. Both have their own velocity:

```
class GeometryObstacle extends GRect {

    constructor() {
        super(OBSTACLES_SIZE, OBSTACLES_SIZE);
        this.vx = 0;
        this.vy = 0;
    }

    move() {
        super.move(this.vx, this.vy);
    }
}
```

Let's continue with our game, we start with the setup(): we initialize the runner and the obstacles. We place the runner in the middle and the obstacles at the bottom of the screen, at a random x-positions. And we must not forget to add the KeyListener.

The game loop is very simple again:

```
function draw() {
    moveObstacles();
    moveDash();
    checkForCollision();
    update();
}
```

The obstacles move with constant speed from right to left, and on the runner only gravity acts. The checkForCollision() method must ensure that obstacles that disappear on the left, appear on the right again, and it should also detect collisions between our runner and the obstacles.

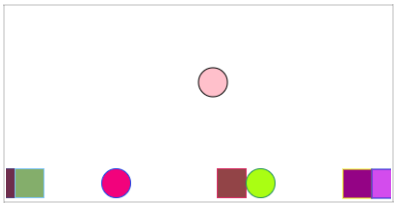
Remains the keyPressed() method: whenever the spacebar is pressed, the y-speed of the runner should get a small push:

```
function keyPressed() {
    if (keyCode == 32) {
        runner.vy -= DASH_JUMP;
        runner.move();
    }
}
```

That's it.

.

JumpAndRun



Try it

GeometryRun is a typical jump-and-run game. Donkey Kong [13] was one of the first known games of this genre, also known as "platform game" [12]. What makes jump and run games different is that there are different types of objects and that there are different levels.

In our JumpAndRun project we need the following global variables:

```
let ball;
let movingObject = new Array(Math.trunc(400 / BALL_DIAM + 1));
```

We have a ball for the player and the movingObjects are an array of GObjects. These could be GOvals, GRects, or any other GObjects. We determine the GObject from the string world:

```
let world = "  RRRR  O   RO  OOO  R";
```

If the string contains an 'R', a GRect is to be created at the position, a GOval for an 'O', and a space is nothing. That means we can describe different levels with different world strings. And a level editor would do nothing else but edit this string.

What does the setup() method do? It creates the ball and creates the world. Here the createNewObjects() method may be helpful:

```
function createNewObjects() {
    for (let i = 0; i < movingObject.length; i++) {
        switch (world.charAt(i)) {
            case 'R':
                let rect = new GRect(APP_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;
            }
        }
    }
```

The game loop is identical to the one in the last project. What is different is the checkForCollisionWithObjects() method:

```
function checkForCollisionWithObjects() {
    let 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, APP_HEIGHT - 2*BALL_DIAM - BALL_OFFSET);
        } else {
            alive = false;
        }
    }
}
```

because depending on the type of object, different things should happen: we can stand on GRects, but when we come in contact with GOvals, we die.

.

MinesClone

Try it

We've all played MineSweeper (or Mines) before. The game is about finding hidden mines without detonating them [14].

1. Playing Field

The first question we have to ask ourselves: how do we want to represent the playing field? One possibility is a two-dimensional array:

```
let field = makeArray(FIELD_SIZE, FIELD_SIZE);
```

where we use the following function to create the array, basically it is an array of arrays:

```
function makeArray(d1, d2) {
    var arr = [];
    for (let i = 0; i < d2; i++) {
        arr.push(new Array(d1));
    }
    return arr;
}
```

In the *initialzeField()* method we initialize this array with strings representing the content of each cell. For instance, an 'M' could represent a mine and a space ' ' means that the cell is empty:

```
function initialzeField() {
    let rgen = new RandomGenerator();
    for (let i = 0; i < MinesConstant.NUMBER_OF_MINES; i++) {
        let x = rgen.nextInt(0, MinesConstant.FIELD_SIZE - 1);
        let y = rgen.nextInt(0, MinesConstant.FIELD_SIZE - 1);
        field[x][y] = 'M';
    }
}
```

2. Mines in the Area

After we have hidden the mines, we have to count how many mines are in the respective environment of a cell. Of course we can do that ourselves, but we may also use the following method:

```
MinesHelper.countMines(field);
```

This method takes our field array as parameter and modifies it. This is possible because arrays are passed by references, i.e. in the original. Each cell (except mines) then contains the number corresponding to the number of adjacent mines:

	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

prior

	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

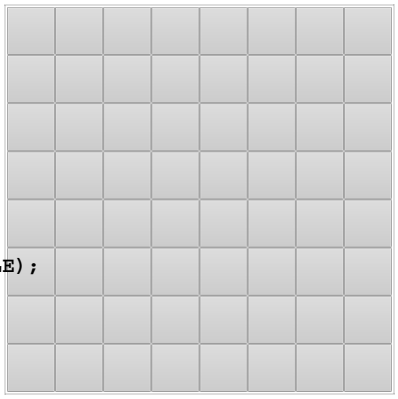
after

3. Display Playing Field

Now that our data structure (the array) is in place, we continue with the graphical part. First, we write the drawInitialField() method. Since in the beginning all cells are still hidden, this is quite simple, we simply draw 8 * 8 "initial.png" images. For this we use the GImage class:

```
img = new GImage("Pr7_Asteroids/initial.png");
add(img, i * PIXEL_PER_TILE, j * PIXEL_PER_TILE);
```

Our game looks already quite similar to the original.



Try it

4. MouseEvents

To be able to react to mouse clicks we need to implement the mouseClicked() method. When the mouse is clicked, the first step is to find out which cell the player clicked on. Here our old friend integer division helps us:

```
let x = Math.trunc(mouseX / MinesConstant.PIXEL_PER_TILE);
let y = Math.trunc(mouseY / MinesConstant.PIXEL_PER_TILE);
```

With these coordinates we can check in our array field[x][y] what is there:

```
if (field[x][y] == 'M') {
    ...
} else if (field[x][y] == '0') {
    ...
} else {
    ...
}
```

If the player clicked on a mine, the game is over. In this case you could write a method drawWholeField() that exposes the entire playing field. Otherwise, we should call the drawOneTile(x, y) method, which draws the correct image for this cell at x,y, i.e. the image of a mine ("mine.png"), if it is a mine, or the image for the empty field ("empty.png") superimposed with a GLabel indicating the number of adjacent mines. If you want, you could also give the GLabel the matching color from the LABEL_COLORS[] array.

5. Marking of Cells

One important aspect is still missing in our MinesClone: the marking of cells as potential mines. In the original this can be done with the right mouse button. This is actually quite simple, because the MouseEvent contains the information which of the mouse buttons was pressed:

```
if (mouseButton === CENTER) {  ...  }
```

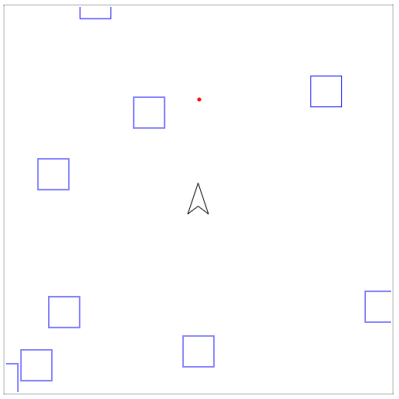
If the player has pressed the third mouse button, i.e. the right one, then the image "marked.png" should be drawn at the corresponding position. As with the previous example, also in MinesClone we want to use an interface (MinesConstant) to store all our constants.

Extensions

You could write a method discoverEmtpyTiles(): if the player clicks on an empty tile, then all empty surrounding tiles could be uncovered.

.

Asteroids



Try it

According to Wikipedia "Asteroids is one of the greatest successes of all time in the history of computer games" [15]. This should not deter us from developing our own version of Asteroids. The game is about flying through an asteroid field with a spaceship. And of course it's about not colliding with the asteroids.

1. Pre-existing Classes

If we want to develop the game in a reasonable amount of time, we may want to use some pre-existing classes:

- **GAsteroid:** is a GRect with speeds vx and vy, as well as a move() method.
- **GBullet:** is a GOval, otherwise identical to GAsteroid.
- **GSpaceShip:** is a GPolygon, just like GAsteroid it has speeds vx and vy, as well as a move() method. Additionally it can spin, rotate(), and accelerate via startEngine().

If we take a closer look at the MarsLander project, then these classes are really nothing new. We include them at the top of our code:

```
include("Pr7_Asteroids/GAsteroid.js");
include("Pr7_Asteroids/GBullet.js");
include("Pr7_Asteroids/GSpaceShip.js");
```

We also need to declare several constants:

```
const FPS = 25;
const APP_WIDTH = 400;
const APP_HEIGHT = 400;
const NR_OF_ASTEROIDS = 10;
const ASTEROID_SIZE = 40;
const ASTEROID_MAX_SPEED = 5;
const SPACE_SHIP_SIZE = 20;
const SPACE_SHIP_SPEED = 3;
const BULLET_SIZE = 4;
const BULLET_SPEED = 10;
```

In the setup() method we initialize the ship and the asteroids. The ship launches from the center of the screen. As for the asteroids, there are supposed to be ten of them, they are blue and randomly distributed. Their speeds should be random:

```
function createAsteroids() {
  for (let i = 0; i < NR_OF_ASTEROIDS; i++) {
    asteroids[i] = new GAsteroid(rgen.nextInt(APP_WIDTH),
      rgen.nextInt(APP_HEIGHT));
    asteroids[i].vx = rgen.nextInt(-ASTEROID_MAX_SPEED,
      ASTEROID_MAX_SPEED);
    asteroids[i].vy = rgen.nextInt(-ASTEROID_MAX_SPEED,
      ASTEROID_MAX_SPEED);
    asteroids[i].setColor(Color.BLUE);
    // asteroid.setFilled(true);
    add(asteroids[i]);
  }
}
```

2. Game Loop

The game loop for Asteroids is only slightly more complicated than our other projects:

```
function draw() {
  if (spaceShip !== undefined) {
    moveSpaceShip();
    moveAsteroids();
    moveBullet();
    checkForCollisions();
  }
  update();
}
```

In the game loop in each iteration we first move the spaceship, then the asteroids, followed by the bullet, if one is fired. And of course we have to check all possible collisions, more about that later.

3. Key Events

The spaceship is controlled by the keyboard, so we have to implement the keyPressed() method:

```
function keyPressed() {
  switch (keyCode) {
    case UP_ARROW:
      spaceShip.startEngine();
      break;
    case 32:
      ...
  }
```

```
}
```

If the player presses the up key (38), then the spaceship should accelerate (startEngine()), if she presses the left key (37), the spaceship should turn 10 degrees to the left, if she presses the right key (39), then the spaceship should turn 10 degrees to the right, that is -10 degrees.

So far so good. If we test our game now, the asteroids should fly around, and our spaceship should be able to spin and accelerate.

What's still missing is our self-defense: if we press the spacebar, the spaceship should fire a bullet. So we need another entry in the keyPressed() method: if the player presses the spacebar, then the method fireBullet() should be called. In fireBullet() we create a new GBullet at the position of the spaceship, and with the following velocities:

```
function fireBullet() {
  if (bullet === undefined) {
    bullet = new GBullet(spaceShip.getX(), spaceShip.getY());
    bullet.vx = Math.sin(spaceShip.angle) * BULLET_SPEED;
    bullet.vy = -Math.cos(spaceShip.angle) * BULLET_SPEED;
    add(bullet);
    sendToBack(bullet);
  }
}
```

A little test should verify that we can fire bullets now. Notice the sendToBack(), that makes sure collision detection works properly.

4. Collisions

The collisions make the game interesting. All in all, there are five different ones:

```
function checkForCollisions() {
  checkForCollisionAsteroidsWithWall();
  checkForCollisionSpaceShipWithWall();
  checkForCollisionBulletWithWall();
  checkForCollisionBulletWithAsteroid();
  checkForCollisionAsteroidWithSpaceShip();
}
```

The collisions with the wall are the easiest. Both the spaceship and the asteroids should simply reappear on the opposite side of the screen when they leave the screen. If the ball leaves the screen, it should simply disappear:

```
removeObj(bullet);
bullet = undefined;
```

To detect collisions between the bullet and an asteroid, we use the getElementAt() method: if there is a GObject where the bullet is, then that must be an asteroid. We then remove the asteroid and the bullet:

```
removeObj(obj);
removeObj(bullet);
bullet = undefined;
```

Very important, we do not set obj to undefined (why?)!

There are still collisions between spaceship and asteroids: for the spaceship they are catastrophic, because they lead to the end of the game. We'll just set the spaceship to null,

```
removeObj(obj);
removeObj(spaceShip);
spaceShip = undefined;
```

and that ends the game loop.

Extensions

We can think of a lot of extensions to our Asteroids game:

- Game over: we could write a method displayGameOver() that displays a big text (SansSerif-36) in the middle of the screen.
- Hyperspace: the player can also send the spaceship into hyperspace: it then simply reappears at a random location on the screen. Of course, there is a risk of self-destruction if it reappears within an asteroid.
- Prettier asteroids: in the real game, the asteroids are not just GRects, but pretty GPolygons. All we have to do is modify the GAsteroid class so that the asteroids look like those in the real game.
- Splitting asteroids (hard): in the real game the asteroids do not just disappear when they are hit by a bullet, but they halve themselves. The smaller parts then move at different speeds in different directions. This is very hard to do with an arrays, but if you use an ArrayList (next chapter), it is actually not that hard.

.

QUESTIONS

1. Name two characteristics of an object-oriented language.

2. Give an example for inheritance

3. Declare an array of numbers with five elements containing the numbers from 1 to 5.

.

REFERENCES

References from Chapter 2 also form the basis of this chapter. Further details on many of the projects can be found in 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

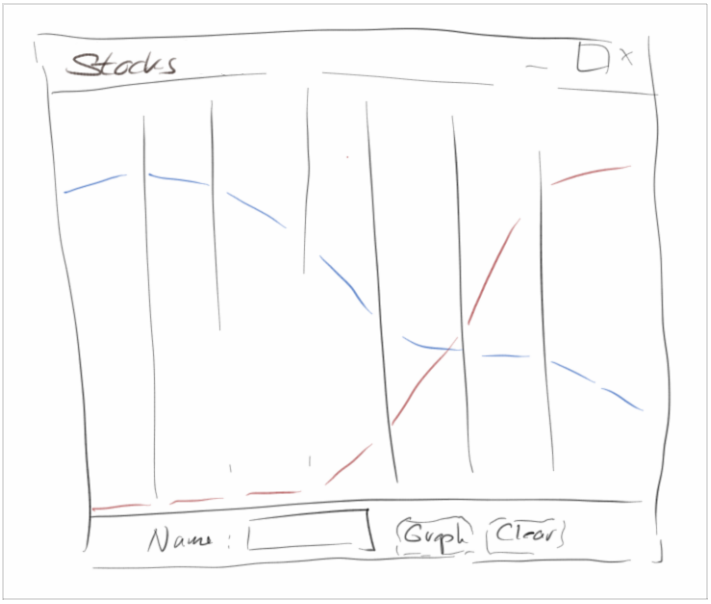
[12] Platform game, https://en.wikipedia.org/wiki/Platform_game

[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

The book is coming to an end, which usually means it's getting interesting. We will cover such topics as files, exception handling and we will get to know our first data structures. We continue with object-oriented analysis, learn a little more about interfaces and hear about polymorphism.

.

Files

As soon as we turn off the computer, all data is gone - unless it has been saved. Saved means that we wrote it to a file. This is easier than you might think, but we start with reading from a file.

To read from a file we have to do three things:

1. open the file,
2. read from the file, line by line, and
3. close the file.

```
Enter file to read
(Ch8_Stocks/TomSawyer.txt):
Ch8_Stocks/TomSawyer.txt
The Project Gutenberg EBook of
The Adventures of Tom Sawyer,
Complete
by Mark Twain (Samuel Clemens)

This eBook is for the use of
anyone anywhere at no cost and
```

Try it

In code it looks like this:

```
// open file
let fr = new Utils.FileReader(fileName);

// read from file, line by line
while (true) {
    let line = fr.readLine();
    if (line == null)
        break;
    println(line);
}

// close file
fr.close();
```

We use our Utils.FileReader class to read text files. We simply pass the file name of the file to be opened.

Then we simply use a Loop-And-A-Half to read line by line from the file using the readLine() method. We know that we have reached the end of the file when readLine() returns null. Then there is nothing more to read, which means

we close the file reader with the close() method.

A brief note on our Utils.FileReader class is in order: a browser can not directly access your file system. What the browser can do, is upload files, we see that later in the project UploadFile. What our class Utils.FileReader does is to load a file from the web server where we host our JavaScript files. Also, interesting, most of the programs we wrote up to now, do work locally from the file system without hosting. The Utils.FileReader does not, because it violate the CORS policy of your browser [8]. And a final remark, why did we not call it simply FileReader? There already exists a class in JavaScript that is called FileReader, which does not really do what you would expect of it, coming from Java. And it is a nice opportunity to show, how to handle naming conflicts in JavaScript.

.

Write to File

When writing to a file we have to do three things:

1. open the file for writing,
2. then write to the file, and
3. close the file.

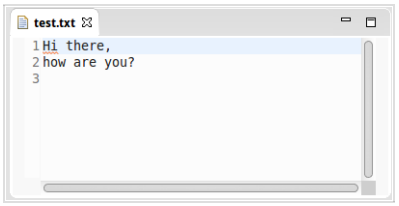
The code looks very similar to the one above:

```
Enter text to write ('.' to quit):
hi there,
this is another test
.
```

Try it

```
// open file
let fw = new Utils.FileWriter("test.txt");

// write to file, one string at a time
println("Enter text to write ('.' to quit): ");
while (true) {
    let line = await readLine("");
    if (line == '.')
        break;
    fw.append(line + "\n");
}

// close file
fw.close();
```

We use the FileWriter class to write to text files. First we pass the file name to which we want to write. Then we use the Loop-And-A-Half again to write line by line using the method write(). The '\n' may also be interesting: it is the character for new line, and ensures that the next string in the "test.txt" file is written to a new line. To avoid getting stuck in an infinite loop, we check if the user has entered a period, this is our abort criterion. Then we close our writer with the close() method.



Like not being allowed to read files, the browser also is not allowed to write files. But what the browser can do is download files. So this is what we do here, we actually download the files, and the user can then choose whether and where to store the files.

.

Data Structures

Data structures are extremely important. Unfortunately, most people stop with the array data structure, which we have seen already. More important, however, is the Map data structure. JavaScript has the Array class and the Map class for these. The equivalent classes in Java are called ArrayList and HashMap. Let us briefly compare the two:

Array	**ArrayList**	**Map**	**HashMap**
new Array()	new ArrayList()	new Map()	new HashMap()
push(value)	add(value)	set(key, value)	put(key, value)

at(idx)	get(idx)	get(key)	get(key)
includes(value)	contains(value)	has(key)	containsKey(key)
indexOf(value)	indexOf(value)	keys()	keySet()
	values()	values()	values()
length	size()	size	size()
splice(indexOf(value),1)	remove(value)	delete(key)	remove(key)

As you can see they are basically equivalent from a user point of view. I wrote little wrapper classes in the utils library, so you can choose either the Java syntax or the JavaScript syntax. The simple reason for this is, that I had all the examples written in Java, and did not want to do a complete rewrite. I was surprised that this worked so well, that I did want to show you this, because it is a very nice example of software portability [9].

.

ArrayList

Let's start by declaring and instantiating an ArrayList:

```
let names = new ArrayList();
```

Here we create an ArrayList with the name "names". An ArrayList can grow and shrink, depending on requirements. That's very practical!

Next, we want to add a few names to our list of names, but only if the name is not already in the list:

```
while (true) {
    let name = await readLine("Enter new name: ");
    if (name.length == '')
        break;
    if (!names.contains(name)) {
        names.add(name);
    }
}
```

```
Enter new name: fritz
Enter new name: lisa
Enter new name: lisa
Enter new name: hans
Enter new name: fritz
Enter new name:

Unique names:
fritz
lisa
hans
```

Try it

With the method contains() we check, if a certain name is already in the list. If not, we add it with add(). Also, we don't have to specify an index when adding, new entries are simply inserted at the end of the list.

If we want to print our list of names, we have to iterate over the list:

```
for (let i = 0; i < names.size(); i++) {
    println(names.get(i));
}
```

With the size() method we can determine how many entries our list has, and with get() we read the entry at a certain position. Additionally, the ArrayList has the following practical methods:

- **set():** replaces the entry at a certain position with a new one,
- **indexOf():** searches for an entry and returns the position,
- **remove():** removes an entry from the list, and
- **clear():** deletes the complete list.

But as I stated before, you can also use the equivalent JavaScript Array methods.

.

.

.

HashMap

HashMaps are probably the most useful of all data structures: because we can use them for searching. Classic examples of hash maps are telephone books and dictionaries. Let's take a closer look at the former, the PhoneBook. We'll start with the HashMap declaration:

```
let phoneBook = new HashMap();
```

A HashMap associates a value with a key, which is why it is sometimes also called an associative array.

Let's add some values to our PhoneBook:

```
while (true) {
    name = await readLine("Enter name: ");
    if (name == '')
        break;
    let number = await readInt("Enter number: ");
    phoneBook.put(name, number);
}
```

```
Fill phone book with data
Enter name: fritz
Enter number: 123456
Enter name: lisa
Enter number: 222222
Enter name: hans
Enter number: 654654
Enter name:

Search phone book
Enter name to search: lisa
222222
Enter name to search: heinz
no entry for this name
Enter name to search:

All phone book entries
fritz: 123456
lisa: 222222
hans: 654654
```

Try it

Inserting is done with the method put(): it takes two arguments, a name and a number. If the name already exists in the map, it is overwritten, if not, it is inserted. In a HashMap there can never be two entries with the same name, keys must be unique.

After we have filled our PhoneBook with some data, we want to search in it, because that's what the HashMap is good for:

```
name = await readLine("Enter name to search: ");
if (name == '')
    break;
if (phoneBook.containsKey(name)) {
    println(phoneBook.get(name));
} else {
    println("no entry for this name");
}
```

With the containsKey() method we can test if a name exists in the map, and with get() we can get the number to the name. This looks very similar to the ArrayList, but it is much cooler: with the ArrayList we have to know the index of the entry we want to access, with the HashMap we simply say: give me the phone number for that name.

Just in case we want to list all phone numbers in our PhoneBook, let us iterate through our map:

```
for (const name of phoneBook.keySet()) {
    let number = phoneBook.get(name);
    println(name + ": " + number);
}
```

This looks different: the first line seems to make no sense at all. The problem with maps is that they do not have an index, i.e. we can not say give me the second element, because a map does not know what its second element is (or at least it doesn't tell us). There is also not really any fixed ordering, i.e. the order in which the entries are output may be completely different from the order in which they were inserted.

This is why the iteration uses this pseudo-for-loop trick:

```
for (const name of phoneBook.keySet()) { ... }
```

It basically says: give me a name from the list of names in the map. And after that one, you give me the next one,

etc. until we're all through. Once you've accepted that is how a map works, the rest is easy.

.

Object-Orientierted Analysis

We are at a transition: we are just leaving the procedural one-class world and entering the object-oriented multi-class world. This transition is not entirely painless. And where the top-down approach has served us really well so far, it now only helps us on a small scale, but no longer on a larger scale.

However, this is where object-oriented analysis comes to the rescue. Before we start to write any code, we first need to think about what we want. This is called requirements analysis. It is best to write the requirements down in very simple sentences. For example, the requirements for a web shop could be formulated in the following way:

> "The Azamon Shop has articles and shopping carts. An article has a name, type and price. A shopping cart has a user name and a list of articles. We can list all the articles of the shop. We can list all the articles in a shopping cart. We can put an article in a shopping cart. We can calculate the price of all articles in a shopping cart."

These requirements are the basis for our object-oriented analysis.

1.Step: In the first step of this analysis we take a colored pencil and underline the verbs green and the nouns with red:

> "The Azamon Shop has articles and shopping carts. An article has a name, type and price. A shopping cart has a user name and a list of articles. We can list all the articles of the shop. We can list all the articles in a shopping cart. We can put an article in a shopping cart. We can calculate the price of all articles in a shopping cart."

2.Step: In the second step we make a list for all the verbs:

- has
- list
- list
- put
- calculate

and the nouns are listed and counted:

- shop: II
- article: IIIIIII
- shopping cart: IIIII
- name: I
- type: I
- price: II
- user name: I

Counting is not absolutely necessary, but helps to recognize what may be more important.

3.Step: In the third step we look at the list of nouns. Which of the nouns can we describe with a simple data type like int, double, string etc.?

- shop: ???
- article: ???
- shopping cart: ???
- name: String
- type: String
- price: int
- user name: String

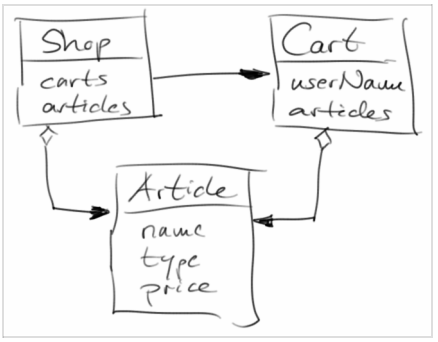
If something is complicated, i.e. consists of other parts, such as shop, article and shopping cart, then it is a complicated data type, i.e. a class. So we have identified our classes: Shop, Article and ShoppingCart.

4.Step: The fourth step is to assign the right attributes to the right classes. To do this, we simply re-read our requirements. They say:

- "The Azamon Shop has articles and shopping carts": so obviously articles and shopping carts belong to the shop, are therefore attributes of the Shop.
- "An article has a name, a type and a price": so name, type and price are part of the Article.
- "A shopping cart has a user name and a list of articles": so user name and list of articles belong to the ShoppingCart.

At this point it also makes sense to pay attention to the plural: every time a word appears in the plural, this means that we need a list (or map) of the respective attributes. For example, "list of articles" or "shopping carts" then become ArrayLists.

With this information we can already write down our classes with attributes:




```
class Shop {
    constructor() {
        this.carts;
        this.articles;
    }
}

class Article {
    constructor() {
        this.name;
        this.type;
        this.price;
    }
}

class AzamonCart {
    constructor() {
        this.userName;
        this.articles;
    }
}
```

5.Step: The last step is to assign the methods to the correct classes. All the verbs we collected in the second step, turn into methods (we have always said that methods are verbs). We go through our list of verbs one at a time:

- has: "has" does not count as a verb, since it expresses affiliation and was already used in step 4.
- list: refers to "We can list all items in the shop", so it is part of the Shop and should be called listArticles().
- list: refers to "We can list all items in a shopping cart", so it is part of the ShoppingCart and should also be called listArticles().
- put: refers to "We can put an item in a shopping cart", therefore it belongs to the ShoppingCart and should be called putArticle() or addArticle().
- calculate: refers to "We can calculate the price of all items in a shopping cart", so it belongs to the ShoppingCart, and should be called calculatePrice().

After this analysis, our classes, their attributes and methods are basically fixed:

```
class Shop {
    constructor() {
        super();
        this.carts;
        this.articles;
    }

    listArticles() {
    }
}

class Article {
    constructor() {
```

```
        super();
        this.name;
        this.type;
        this.price;
    }
}

class Cart {
    constructor() {
        super();
        this.userName;
        this.articles;
    }

    listArticles() {
    }

    addArticleToCart() {
    }

    calculatePriceOfArticlesInCart() {
    }
}
```

What we have not decided yet are the parameters the methods accept and what they return. And it never hurts to give each class a constructor. And of course a minor thing: the code is still missing.

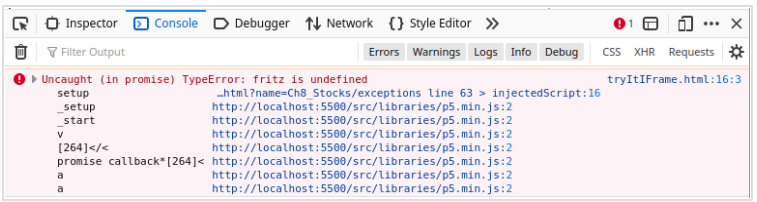
.

Errors

What if something goes wrong? Or first of all, what can go wrong? For example, look at the following two lines of code:

```
let fritz;
fritz.setColor(Color.RED);
```

If you run this in the browser, nothing happens. However, if you look at the browser's console (usually with F12) you will see the following error message:



The error message says that fritz is undefined. The error is a *TypeError*. It even tells you where the problem is: at line 16 column 3 of your JavaScript file (16:3). Usually it provides a link which you can click on and it navigates you automatically to the problematic line in the code.

Can anything else go wrong? Yes, many things can go wrong. Here are a couple of typical cases:

```
// TypeError
let num = 1;
num.toUpperCase();

// RangeError
let b = new Array(-1)

// ReferenceError
```

```
let x = 3;
x = x + y;

// SyntaxError
eval("let s = 'hi");
```

You want the upper case version of a number, or an array with a negative size, or you are using a variable that has not been initialized, and then there can be errors in you JavaScript code itself.

What happens to our program when an error occurs? Well, other programming languages just crash., but JavaScript is very cool about this: it tries to ignore the errors, and continues as if nothing had happened, kind of like my mother in law. But naturally, the results that come out will most likely be wrong or nonsense. Hence, what we should do is appropriate error handling with try-catch.

.

Try-Catch

With a try-catch statement we first try if everything works. If it works, fine, if not, then we tell the program what to do. In code it looks like this:

```
try {
    // problematic code
    ...
} catch ( e ) {
    // deal with error
    ...
}
...
```

```
TypeError: Cannot read properties
of undefined (reading 'setColor')
RangeError: Invalid array length
ReferenceError: y is not defined
SyntaxError: Invalid or
unexpected token
Infinity
NaN
3
```

Try it

We surround our problematic code with a try block. If everything works, the program continues normally (the catch block is not executed). If something goes wrong, however, the program does not terminate, but jumps to the catch block instead and executes it. After that, the program assumes that we have everything under control and executes the code after the catch block as if nothing had happened.

Interesting maybe that what would be errors in other languages, are no problem for JavaScript:

```
// ArithmeticException
let x = 5 / 0;
console.log(x);        // Infinity

// NumberFormatException
let y = parseInt("five");
console.log(y);        // NaN

// ArrayIndexOutOfBoundsException
let eggs = [0, 1, 2, 3];
console.log(eggs[5]);  // undefined
```

throw

If you want to, you can also throw your own errors. For instance,

```
if (!locateKarel()) {
    throw Error("Could not locate Karel in the world");
}
```

would throw an error, which then should be caught with some try-catch.

"use strict"

While we are at errors: recall that in JavaScript we are not required to declare variables. This can lead to problems:

```
//"use strict";  // uncomment this line

function run(() {
    let fritz = 5;
    frits = 4;
    console.log(fritz);
}
```

In the third line, we want to reassign a new value to the variable *fritz*. But we made a typo, and hence instead of a reassignment, we declared a new variable named *frits*. JavaScript does not care, there is no error here. To avoid these kinds of stupid mistakes, it is highly recommended that you start all of your JavaScript code with the "use strict" keyword. Adding it, will result in an error being thrown.

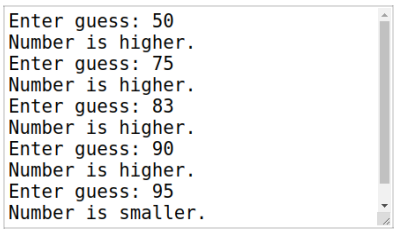
SEP: Always add the line "use strict" at the top of your code!

.

Interfaces

Finally we come to the last topic of this chapter: *interfaces*. We could make this very short by saying that JavaScript does not have interfaces or abstract classes. Period.

But obviously, if so many languages have it, there must be a good reason for it. Until now we have always programmed alone. But what if we work in teams, meaning several people write on the same program? How do we know who writes what? How can we be sure the different pieces fit together? We use interfaces for exactly this purpose.



Try it

Let's take NumberGuess as an example. The objective of the game is for the computer to choose a random number, let's say between 0 and 99, and for the player to guess that number in as few steps as possible. The program consists of two parts: the part that takes care of the logic (NumberGuessLogic) and the part that interacts with the player (NumberGuessConsole). This means we have two classes, who depend on each other. We want to clearly define this dependency, and we do this by means of an interface. Since JavaScript does not have interfaces or abstract classes, we just use a normal class:

```
// interface or abstract class
class NumberGuessLogic {

    constructor() {
        // if we need 'this' in constructor of subclass, then super() must be called,
        // hence we can not throw an Error here:
        // throw new Error('NumberGuessLogic is interface, can not be instantiated!');
    }

    /**
     * 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
     *
     * @param guess
     * @return
     */
    makeGuess(guess) {
        throw new Error('Method is abstract, needs to be implemented!');
    }
}
```

This interface is a contract between two developers:

- For one developer (NumberGuessConsole) it means that there will be a method *makeGuess(int guess)*, which takes the number guessed as a parameter, and which returns whether the number was to small, to large or correct (-1, +1, 0).
- For the other developer (NumberGuessLogic) it means that she has to write a method *makeGuess(int guess)*, which compares a guess with the internally generated random number and returns whether the number was to small, to large or correct (-1, +1, 0).

This allows two or more developers to independently write separate pieces for a common program that can then easily be merged. Notice two things about the code: it has lots of comments, because two people have to agree explicitely what the method is supposed to do, there should be no ambiguity. And it throws an error, indicating, that this class is not really supposed to be used.

Let's continue with how this would be used: We start with the part that implements the NumberGuessLogic interface. It is customary to add the extension "Impl" to classes that implement a given interface, hence the name NumberGuessLogicImpl:

```
// implementation of interface
class NumberGuessLogicImpl extends NumberGuessLogic {

    constructor() {
        super();
        const rgen = new RandomGenerator();
        this.number = rgen.nextInt(0, 99);
    }

    /**
     * guess should be number between 0 and 99 return 0 if guess was correct
     * return +1 if guess was higher return -1 if guess was lower
     *
     * @param guess
     * @return
     */
    makeGuess(guess) {
        if (guess == this.number) {
            return 0;
        }
        if (guess > this.number) {
            return +1;
        } else {
            return -1;
        }
    }
}
```

This class must have a method called makeGuess(int guess), which of course must do something now. Naturally, the random number must be generated somewhere, probably in the constructor.

Regardless of this, the other developer can already start with NumberGuessConsole class. This is a normal console program:

```
async function setup() {
    createConsole();

    // let logic = new NumberGuessLogic();
    let logic = new NumberGuessLogicImpl();

    if (logic instanceof NumberGuessLogic) {
        while (true) {
            let guess = await readInt("Enter guess: ");
            if (logic.makeGuess(guess) == 0)
```

```
                break;
            if (logic.makeGuess(guess) == 1) {
                println("Number is smaller.");
            } else {
                println("Number is higher.");
            }

        }
        println("You won!");
    } else {
        println("'logic' is not of type NumberGuessLogic!");
    }
}
```

Notice first that with the *instanceof* operator we make sure that the Impl class is of the correct type. And second the commented line, that is the line the second developer used before the Impl class was finished, to write her code. When both developers are done, we just put the classes together into one project, and everything should work fine.

We could have used the same approach also in the TicTacToe and the MindReader projects, if we wanted to.

.

Polymorphism

Steve Jobs used to end his presentation with the sentence "one more thing..." and usually that's when it got interesting. Polymorphism literally means "many forms". And what does this have to do with JavaScript?

```
Freshman [name=Hänschen, id=-1, credits=0]
Student [name=Gretchen, id=-1, credits=0]
```

Try it

Let's remember our Student example from the MindReader chapter. We defined a class Student:

```
class Student {
    constructor(name, id = -1) {
        this.name = name;
        ...
    }

    toString() {
        return "Student [name=" + this.name + ", id=" + this.getId()
      + ", credits=" + this.incrementCredits() + "]";
    }
}
```

In addition, we could also define a class Freshman, which extends a Student:

```
class Freshman extends Student {
        constructor(name, id = -1) {
                super(name, id);
        }

    toString() {
        return "Freshman [name=" + this.name + ", id=" + this.getId()
      + ", credits=" + this.incrementCredits() + "]";
    }
}
```

This means, Freshman can do everything Student can, the only difference is when calling the toString() method.

Since both classes are of type Student, we can assume that they have a toString() method, and thus the following function makes totally sense,

```
function polymorphic(studnt) {
    println(studnt.toString());
}
```

if it is called with with a Student or a Freshman object:

```
async function setup() {
    createConsole();

    const hansel = new Freshman("Hänschen");
    polymorphic(hansel);
    const gretel = new Student("Gretchen");
    polymorphic(gretel);
}
```

Now in JavaScript there is a little problem: there is no compiler preventing us from doing stupid things. For instance we could call the polymorphic() function with a string:

```
    polymorphic("hi there");
```

That would work, but it is not really what we wanted. But if we change the polymorphic() function slightly,

```
function polymorphic(studnt) {
    if (studnt instanceof Student) {
        println(studnt.toString());
    } else {
        throw Error("Parameter is not of type Student!");
    }
}
```

we get exactly what we wanted: a method that works for both Students and Freshmen, meaning it is polymorph. Well, as polymorph as it is going to get in JavaScript.

We now know all three pillars of object-oriented programming:

- Data Encapsulation and Information Hiding,
- Inheritance and Composition,
- and Polymorphism.

.

.

Review

It was about time we learned how to work with files. Also the exception handling should have come much earlier, but of course it makes much more sense to introduce it together with files. What is really new and will prove very useful for the future are the data structures ArrayList and HashMap. They will make our lives easier many times over. And we have checked off the topics of interfaces and polymorphism, nothing really to be afraid of.

But the most important thing in this chapter was the object-oriented analysis, so to speak Top-Down 2.0.

.

Projects

Was all the trouble worthwhile? Well, see for yourself.

.

WriteToStorage

Most browsers have something that is called local storage. It kind of acts like a file system, and one way we can use it is as a kind of replacement. The API is very similar to the FileWriter's:

```
async function setup() {
    ...

    let fw = new StorageWriter('test.txt');
    fw.clear();

    println('Enter text to write (\'.\' to quit): ');
    while (true) {
        let line = await readLine('');
        if (line == '.')
            break;
        fw.append(line + '\n');
    }
}
```

```
Enter text to write ('.' to quit):
hi there
this is a test!
.
```

Try it

Naturally, you must allow access to local storage in your browser for this to work.

.

ReadFromStorage

Once you have written something, maybe you also want to read it, that's what the StorageReader is for:

```
async function setup() {
    createConsole();

    let fileName = await readLine("Enter file to read (test.txt): ");

    // open file
    let sr = new StorageReader(fileName);

    let line = sr.read();
    println(line);
}
```

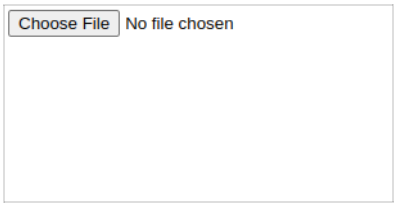
```
Enter file to read (test.txt):
test.txt
hi there
this is a test!
```

Try it

You can't really store a lot in the storage, but it is easy to use and sometime comes in quite handy.

.

UploadFile



Try it

When we were discussing the FileReader, we mentioned that the browser does not really have access to the local file system of the user. But, what we can do in a browser is upload files. In this case, the user usually has to select a file on her computer, and then click on an upload button. The class JSFileUpload wraps the underlying HTML <input type='file'> tag:

```
function setup() {
    createGUI(300, 150);

    let fritz = new JSFileUpload();
    addWidget(fritz);
}
```

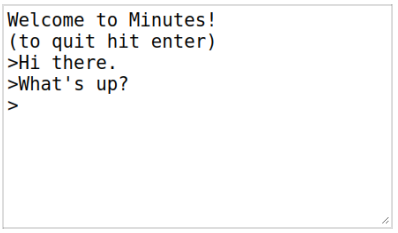
Once the user selected a file, we can read it in the onChange() function:

```
function onChange(ev) {
    let input = ev.target;
    let reader = new FileReader();
    reader.onload = function () {
        let text = reader.result;
        print(text);
    };
    reader.readAsText(input.files[0]);
}
```

Notice, that the FileReader class used here is JavaScript normal FileReader class, and not our Utils.FileReader class.

.

Minutes

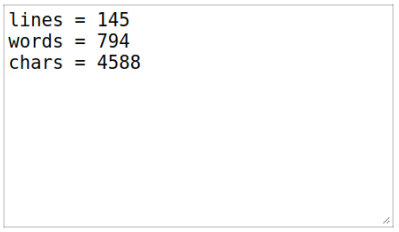


Try it

Sometimes we are responsible for the minutes of a team meeting. Wouldn't it be nice to have a program for taking the meeting notes? The idea is this: just type a sentence, press Enter and the line will be saved in a file called "minutes.txt". For this we open a file with a FileWriter, read line by line (loop-and-a-half) from the console with readLine(), and write each line into the file immediately. Of course, we still need an abort criterion (sentinel): this could simply be an empty line. After that, of course, we must not forget to close our file. A useful extension would be to add the time when the text was entered to each line. Then our program would really deserve its name.

.

WordCount



Try it

WordCount is a very simple program: it reads a specific file line by line and counts the lines, words and characters it contains. We use the FileReader together with the BufferedReader to read line by line. We need three variables for counting:

```
let counterLines = 0;
let counterWords = 0;
let counterChars = 0;
```

(these can be local variables). Counting the lines is very easy. Counting the characters is also easy, because we know how many characters are in a line with line.length(). To count the words we could either use the StringTokenizer, or use the method split() of the String class:

```
function countWords(line) {
    let words = line.split(" ");
    return words.length;
}
```

The split() method should be used with caution. Apparently it looks as if it would simply cut a string into its individual parts and store these individual parts in a string array. Sure it does that, but what criterion does it use to cut the string? Unlike the StringTokenizer, which simply takes a list of separators to split a string, the split() method uses regular expressions. So until we know what regular expressions are we shouldn't really use the split() method.

.

CreateFileWithRandomNumbers

From time to time (e.g. in the next project) we need a file with some random values. These could be just numbers, but these could also be random names or addresses or whatever. The user should tell us the name of the file into which we are supposed to write the random numbers, then we need the range in which the numbers should be, and we need to know how many numbers we should generate. The program should then create a file using FileWriter and RandomGenerator. For the next project we need those random numbers.

```
Create file with random numbers
-------------------------------
Enter file name: Scores.txt
Enter lowest number: 0
Enter highest number: 100
How many numbers do you want: 43
```

Try it

.

Histogram

To get a quick overview of the grade distribution in an exam, it would be nice to have a histogram of that data. You can of course do this with some fancy graphics program (comes later), but it's much quicker to do this with a console program.

We want to accumulate points, so we count how many exams had points between 0 and 9, between 10 and 19, and so on. For example, we could define an array for eleven numbers, and we want the array to be prefilled with all zeros:

```
00-09: ***
10-19: **
20-29: *******
30-39: ***
40-49: *
50-59: ****
60-69: ******
70-79: ****
80-89: ****
90-99: *******
100  : **
```

Try it

```
let histogramData = new Array(11).fill(0);
```

Then there is a small trick on how to easily add entries into the array using integer numbers:

```
function putScoreInHistogram(score) {
    const idx = Math.trunc(score / 10);
    histogramData[idx]++;
}
```

That's a tough line to digest, but after long enough admiration, let's move on.

We read line by line from our Scores.txt file, convert the strings into ints using

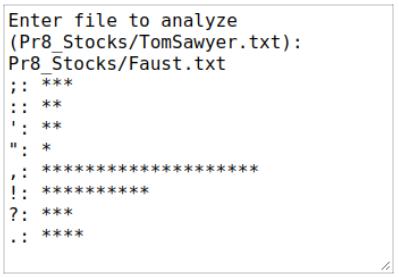
```
let score = parseInt(line);
```

Then we add them to our array using our miracle method putScoreInHistogram(). After reading all our data, we iterate through our array and output it to the console. We could just print the numbers, but much prettier are little stars (also called asterisks not Asterix!):

```
function convertToStars(i) {
    let stars = "";
    for (let j = 0; j < i; j++) {
        stars += "*";
    }
    return stars;
}
```

.

Punctuation



Try it

When we browse the Internet, there are programs (or rather algorithms) that can automatically recognize in which language we are writing. We type a few words and punctuation marks, and the program can tell us what language we are using. Actually, it turns out this is not really that difficult.

The idea in this project is to count the punctuation marks, such as: ";,.!?'. Similar as in the last project (histogram) we read a given text from a file and then show the frequency of punctuation marks in a histogram. It turns out that each language has its favorite punctuation marks, and we can recognize a language by the frequency with which certain punctuation marks occur. Of course the text to be analyzed should be of a certain length, and it should be typical for a given languag (so Ulysses would be rather unsuitable [2]).

The highlight of this program is the instance variable punctuation:

```
const punctuation = ";:'\",!?.";
```

i.e. a string containing all possible punctuation marks. We have specially marked the character '\"' here, because it is the only way to get the quotation mark into a string.

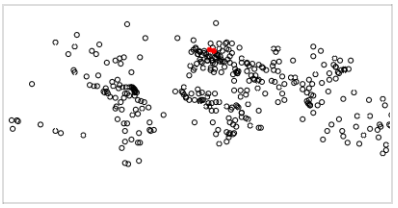
We then read line by line from a file that the user gave us, and analyze each line:

```
function analyzeForPunctuation(line) {
    for (let i = 0; i < line.length; i++) {
        let c = line.charAt(i);
        if (punctuation.includes(c)) {
            let index = punctuation.indexOf(c);
            histogramData[index]++;
            totalNrOfPunctuations++;
        }
    }
}
```

That's pretty heavy tobacco. Again, we read line by line. Then we iterate over each character in each line, one character at a time. We check if it is a punctuation mark, and if so, we increase the counter for that punctuation mark by one. We also have a counter for the total number of punctuation marks, which allows us to normalize the numbers when displaying the asterisks.

.

WorldMap



Try it

Wikipedia has a list with the longitude and latitude of many cities worldwide [3]. If we go through this list, "Cities.txt", and paint a GOval for each city, then we can draw a world map. Again, we use the FileReader/BufferedReader combo to read line by line from the file. The data is in the following form:

```
Germany, Berlin, 52, 32, N, 13, 25, E
```

We must parse, i.e. transform this information in such a way that it is useful for us. For our purposes, the country and the name of the city are useless. Also, we do not need a very high accuracy, therefore the degree specification, i.e. the 52, suffices for us. What we do need to know is whether the city is in the northern or the southern hemisphere:

```
// Germany, Berlin, 52", 32', N, 13", 25', E
let data = line.split(",");
let country = data[0].trim();
let name = data[1].trim();
let lat1 = data[2].trim();
let lat2 = data[3].trim();
let lat3 = data[4].trim();

let lat = parseInt(lat1);
if (lat3.endsWith("S")) {
    lat = -lat;
}
...
```

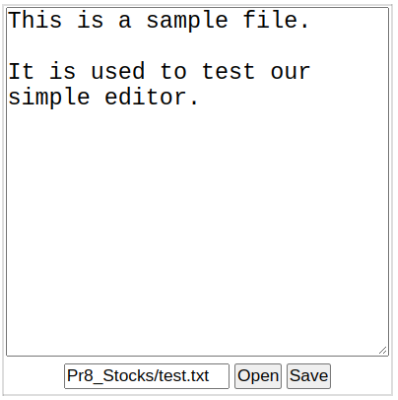
We do the same for length (longitude). We scale our values to fit on our screen, hence

```
let x = (0.5 - lon / 360.0) * WIDTH;
let y = (0.5 - lat / 180.0) * HEIGHT;
let city = new GOval(CITY_SIZE, CITY_SIZE);
add(city, x, y);
```

And that's it. Looks very pretty, doesn't it?

.

Editor



Try it

In the Swing chapter we already wrote a UI for an editor. Now finally we can read and write files. In the saveFile() method we simply get the text from the JSTextArea display using

```
let text = display.getText()
```

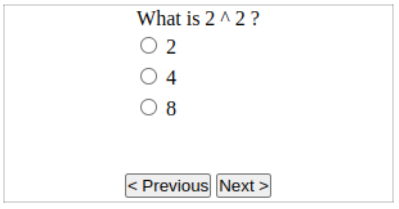
and write the whole text to a file using a FileWriter. In the method openFile() we read line by line from the file into a string and then display it in the JSTextArea display using display.setText(text).

.

.

.

Quiz



Try it

Also for the quiz program we have written the UI already. However, a quiz with only one question is not very useful. We want to be able to do various quizzes with many questions.

We'll start with the questions. If they come from a text file, then we can give any kind of quiz. For our example, this is simply a text file called "Quiz.txt":

```
Correct:  1 + 1 = 2 ?; Yes; No; Maybe
```

```
What is 2 ^ 2 ?; 2; 4; 8
A zebra has stripes?; Yes; No
```

Each line corresponds to a question with the possible answers. The question comes first, followed by the possible answers. Question and answers are separated by semicolons. We can read this file line by line, and using the split() method we get questions and answers:

```
let words = line.split(";");
```

Each line we turn into a Question object,

```
class Question {
    constructor(words) {
        this.question = words[0];
        this.answers = words;
    }

    toString() {
        return this.question + this.answers;
    }
}
```

simply by passing this array to the constructor. Since the Question class is only used in our quiz class, it can be a local class, but does not have to be.

Now what do we do with the questions? We put all of them into a global ArrayList:

```
let questions = new ArrayList();
```

In summary, this is what the parsing part looks like:

```
while (true) {
    let line = fr.readLine();
    if (line == null)
        break;
    let words = line.split(";");
    let q = new Question(words);
    questions.add(q);
}
```

Let's get to the presentation part of our program. Of course we need an actionPerformed() method for the Previous and Next JSButtons:

```
function actionPerformed(e) {
    if (e.getSource() == btnNext) {
        currentQuestion++;
        currentQuestion = currentQuestion % questions.size();
        setQuestion(currentQuestion);
    } else if (e.getSource() == btnPrevious) {
        ...
    } else {
        ...
    }
}
```

For this we need a counter, currentQuestion, which points to the current question, i.e. the index in the ArrayList questions. We still need to implement the setQuestion() method:

```
function setQuestion(index) {
    let q = questions.get(index);
    lbl.setText(q.question);
    buildMultipleChoiceAnswers(q.answers);
}
```

It simply gets the current question from the ArrayList, sets the label to the question and calls our old buildMultipleChoiceAnswers() method with the answers.

Extensions: What is still missing is that the user's answers are stored somewhere and graded at the end. This is not difficult, but somebody has to write the code.

.

Dictionary

Dictionaries are a typical application for a HashMap. As an example we want to implement a German-English dictionary. We hold the dictionary as an instance variable:

```
Enter german word: hund
dog
Enter german word: katze
cat
Enter german word:
```

Try it

```
let dictionary = new HashMap();
```

and the first step is to initialize the dictionary:

```
function initialzeDictionary() {
    dictionary.put("hund", "dog");
    dictionary.put("katze", "cat");
    dictionary.put("fisch", "fish");
}
```

We then ask the user for a German word using readLine() from the console and with the get() method,

```
let english = dictionary.get(german.toLowerCase());
```

we get the English translation from the map. Very important, the translation only goes in one direction.

.

StateLookup

Another typical application for HashMaps are lookup tables: for example, we have the abbreviation of a US state (e.g. "NY") and we would like to know which state it is. Looking on the internet we can find tables with the list of all states, e.g. of the form:

```
Enter state initial (e.g. NY): NY
New York
Enter state initial (e.g. NY): IA
Iowa
Enter state initial (e.g. NY):
```

Try it

```
AL,Alabama
AK,Alaska
AZ,Arizona
...
```

So we read line by line, and insert them into our HashMap:

```
function readStateEntry(line) {
    let comma = line.indexOf(",");
    let stateInitial = line.substring(0, comma).trim();
    let stateName = line.substring(comma + 1).trim();
    states.put(stateInitial, stateName);
}
```

(you can also do this with the StringTokenizer or the split() method of the String class). The rest works in the same way as in the previous project.

.

VocabularyTrainer

```
What is the translation of dog?
Katze
sorry, not correct
What is the translation of cat?
Katze
good job!
What is the translation of fish?
```

Try it

Related to a dictionary is a vocabulary trainer. However, our vocabulary trainer should work for any language, so we can't just hard-code the vocabulary, but should read it from a file. The words are stored in the file "Vocabulary.txt" in the following form:

```
dog,Hund
cat,Katze
fish,Fisch
...
```

We read line by line and put the word pairs in a HashMap called vocabulary:

```
let vocabulary = new HashMap();
```

What distinguishes the vocabulary trainer from the dictionary is that it asks the user for the translation of an English word. If the user's answer is correct, he gets a verbal pat on the back. However, if the user does not know the answer, we have to save the word in a list of unknownWords:

```
let unknownWords = new ArrayList();
```

At the end we should list all the words the user did not know, so he can study them some more later.

.

VocabularyTrainerSwing

Try it

Console applications are simple, but ugly. Since we did all the hard work already in the last project, let's turn it into a UI application. The application should consist of a JSLabel in the north and a JSTextField in the south. We show the user the word to be translated in the JSLabel, and the user should then enter the translation in the JSTextField.

We would like to use a dialog box for feedback to the user. In Swing this is possible with the JSOptionPane:

```
function actionPerformed(e) {
    let english = englishLbl.getText();
    let guess = germanTf.getText();
    if (guess.toLowerCase() == dictionary.get(english)) {
        JSOptionPane.showMessageDialog(this, "Great job!",
            "Check", JSOptionPane.INFORMATION_MESSAGE);
    } else {
        JSOptionPane.showMessageDialog(this, "Try again!",
            "Check", JSOptionPane.INFORMATION_MESSAGE);
    }

    setRandomWord();
}
```

Otherwise, we can reuse the code from the project VocabularyTrainer.

.

Trains

```
Available cities are:
  Nuremberg, Munich, Hamburg,
Berlin, Frankfurt,
Where do you want to start:
Hamburg
From Hamburg you can go to:
  Berlin,
Where do you want to go next:
Berlin
From Berlin you can go to:
  Hamburg, Nuremberg,
Where do you want to go next:
```

Try it

A very nice application for HashMaps are timetables for trains. Let's assume we want to go from Munich to Berlin and there is no direct connection. Then we check the timetable and see that you can travel from Munich to Nuremberg, and from Nuremberg there is a train to Berlin. A timetable could be a text file containing all the connections in the following form:

```
Nuremberg > Berlin
Nuremberg > Frankfurt
Nuremberg > Munich
Munich > Nuremberg
Hamburg > Berlin
```

It is important here that the connections between cities have a direction, so the train goes from Nuremberg to Berlin, but does not have to go back.

The next step is to think about how to put such a timetable in a HashMap. It is clear that the key must be the starting station. But since a key in a HashMap must be unique, this means we can not have two or more values for one key. However, there is a solution: we simply use a list for the destination stations:

```
let connections = new HashMap();
```

At first this may seem a little difficult to understand: what we have here is a hash map that has a string as key but a list as values.

In addition, it also makes sense to have a list of all stations:

```
let cities = new ArrayList();
```

In the setup() we read the timetable and fill our two data structures. With the StringTokenizer we separate source from destination:

```
let st = new StringTokenizer(line, ">");
let source = st.nextToken().trim();
let destination = st.nextToken().trim();
```

Then we should check if the departure station already exists.

```
if (!cities.contains(source)) {
    cities.add(source);
    connections.put(source, new ArrayList());
}
```

and finally we have to add the new connection:

```
let cits = connections.get(source);
cits.add(destination);
```

Now that the data is loaded, we can continue with the actual program. First, we should give the user a list of all departure stations. From this she should choose her starting station. In the next step we list the possible destination stations and let the user choose again. We do this until the user has reached her destination, i.e. enters an empty string.

Extensions: What would be cool, of course, if the user could just enter her starting station and destination station and the program would then automatically suggest a route. Towards the end of next semester we will learn how to do that.

.

Adventure

```
You are currently in kitchen.
You can go to:
ArrayList[hallway,]
Where do you want to go? hallway
You are currently in hallway.
You can go to:
ArrayList[kitchen,living
room,bath room,]
Where do you want to go?living
room
```

Try it

Another nice application for HashMaps are adventure games. Many of these text-based games are about exploring a world and collecting items. We focus here on the exploring part, but the collecting part is also not that difficult.

Similar to the Trains project we need a description of the environment. The easiest way is to describe our apartment. This is what it looks like at home:

```
hallway > kitchen
hallway > living room
hallway > bath room
kitchen > hallway
living room > hallway
bath room > hallway
bath room > kitchen
```

Again we use a HashMap which contains the map of our apartment:

```
let roomMap = new HashMap();
```

Since this is an exploration game, we don't need a list of all rooms, instead we just let the player start exploration in the kitchen. We then list the rooms that can be reached from the kitchen and ask the player to make a choice. In this way, the player can gradually explore our entire home. The game ends when you enter the empty string.

Extensions: There are countless extensions for this game type. One could, for example, depict the world of StarWars or Lord of the Rings in this way. You could hide magical objects in the different rooms. And some rooms can only be entered if you have a certain object, etc...

.

BuildIndex

```
abounding,1
absorbing,3
abundance,1
abundant,1
accepting,1
accessible,1
accident,1
accidental,1
accompanying,2
accomplish,1
```

Try it

Paper books are not as easy to search as electronic books. That's why most of the paper books have an index in the back. As an example we want to create a list of keywords for the book "TomSawyer.txt".

As usual, we walk through the book line by line and use the StringTokenizer

```
let st = new StringTokenizer(line, "[]\"',;:.!?()-/ \t\n\r\f");
```

to extract the words from a line. This is one of the few times where the split() method of the String class would not work (unless you know regular expressions).

So we go through all lines and all words (tokens) and save them in a HashMap:

```
let words = new HashMap();
```

Here the key is the word and the integer counts how often that word occurs in the text. We then fill this HashMap using the following method:

```
function addWordToHashMap(word) {
    if (word != null) {
        if (words.containsKey(word)) {
            let count = words.get(word);
```

```
                words.put(word, ++count);
            } else {
                words.put(word, 1);
            }
        }
    }
```

Now all we have to do is print the map to the console.

Extensions:

- When we look at the list, we find that most words with less than eight letters should not be indexed.
- We could filter out words that end in a plural (this is relatively simple to do in English).
- We could filter out words with useless extensions (like "ly", "ial", "ive", "ous", "ed").
- We can also have a list of stop words and filter those out [6].
- Sorting: if we use a TreeMap instead of a HashMap, the index will be sorted. We'll learn more about sorting in the next semester.

.

Languages

What is a dictionary that translates only from one language into another? We want a dictionary that translates from one language into ten! First we have to find the necessary data somewhere. Fortunately, on the website of the book "Introduction to Programming in Java" by Robert Sedgewick and Kevin Wayne [4] (a great book by the way) there is a file containing the translations of over 800 English words into ten other languages [5].

```
0: English, 1: Danish, 2: Dutch,
3: Finnish, 4: French, 5: German,
6: Indonesian, 7: Italian, 8:
Japanese, 9: Latin, 10: Norwegian,
11: Portugese, 12: Spanish, 13:
Swahili, 14: Swedish,
Enter language: 5
Enter English word: dog
Hund
Enter English word:
```

Try it

The file "Languages.csv" contains the data in the form:

```
"cat","kat","kattekop","kissa","chat, matou, rosse","Katze",...
```

The first word in each line is the English word, followed by the Danish, Dutch, etc., translations. The first line of the file tells us which languages comes at which position. Parsing the data will not be quite as easy as we are used to: when we look at the French translation for cat, we see that there seem to be at least three words for cat in French. However, if we look for quotes using the indexOf() method,

```
function parseLine(line) {
    let translations = new ArrayList();
    while (true) {
        let begin = line.indexOf("\"");
        if (begin < 0)
            break;
        let end = line.indexOf("\"", begin + 1);
        if (end < 0) {
            console.log("***** this should never happen! *****");
        }
        let s = line.substring(begin + 1, end);
        line = line.substring(end + 1);
        translations.add(s);
    }
    return translations;
}
```

parsing is totally feasible. The method parseLine() splits a line from our file and converts it into an array list of strings. This ArrayList contains the English word with all its translations. German is in sixth place, i.e. with

```
let translations = parseLine(line);
let german = translations.get(5);
```

we get the German translation of the word. Thus parsing is done.

Similar to our simple Dictionary project, we want to search for words, and for this we use the HashMap:

```
let dictionary = new HashMap();
```

Since we will not only save one word per English word, but ten, we need to use a list for the values. Filling the list is very simple:

```
let translations = parseLine(line);
dictionary.put(translations.get(0), translations);
```

Translating also is very simple. We just need to know which language is desired (e.g. 5 for German) and the word to be translated:

```
function doTranslation(english, lang) {
    let words = dictionary.get(english);
    if (words != null) {
        return words.get(lang);
    }
    return null;
}
```

If the searched word is in our database, we simply return it, otherwise our method returns null.

Possible extensions could be:

- How could we translate from each of the ten languages into any of the other ten languages?
- Could we translate not just words, but entire sentences?
- Naturally, we could also write a nice UI application for this project.

.

LinesOfCode

In the third semester we will learn about software metrics. The most popular one is Lines of Code (LoC): in earlier days programmers were paid per line of code. It is still a measure of productivity in some companies. So let's write a program that counts how many lines of code we have written. Basically, we have a counter that we increment after reading each line:

```
Pr8_Stocks/languages.js: 88;
[HashMap,Console,FileReader,Array
List,]

/src/Pr8_Stocks/adventure.js: 65;
[Console,HashMap,FileReader,Array
List,]
/src/Pr8_Stocks/azamonShop.js:
50; [ArrayList,]
/src/Pr8_Stocks/buildIndex.js:
```

Try it

```
function countLines(fileName) {
    // open file
    let fr = new Utils.FileReader(fileName);
    let lineCount = 0;

    // read from file, line by line
    while (true) {
        let line = fr.readLine();
        if (line == null)
            break;

        lineCount++;
     }

    fr.close();
    print(fileName + ": " + lineCount + "; ");
}
```

But we can do even more. We can classify the different files according to their contents. For instance, the assume

we are interested in the following keywords:

```
const searchWords = ["Console", "Canvas", "GUI",
    "FileReader", "FileWriter",
    "ActionEvent", "KeyEvent", "MouseEvent", "RandomGenerator",
    "ArrayList", "HashMap", "StringTokenizer", "Color"];
```

then the following function

```
function checkImport(line) {
    for (const word of searchWords) {
        if (line.includes(word)) {
            classification.add(word);
        }
    }
}
```

allows us to tell what topics are covered in that particular file.

.

Library

When we were doing object-oriented analysis, we saw how to create a program from some textual description of what the program is supposed to do, i.e. the requirements. Let's practice this a little more with the university library example:

"The university library has students and books. A student has a name, id and a list of books she has borrowed. A book has an author and a title. A student can borrow books and return books. We can list all books a student has borrowed. We can create new students and we can create new books."

The goal is to determine the classes, attributes and methods of the classes. In order for this to not just stay a theoretical exercise, we want to go one step further and really write the program. Maybe it makes sense to have a quick look at the project InteractiveMenuProgram from chapter 3.

.

Mensa

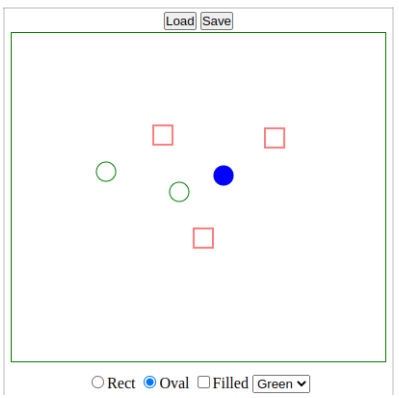
The same thing as above, can also be done for our canteen program:

"The mensa has dishes and ingredients. A dish has a name, a price and a list of ingredients. An ingredient has a name, a price and calories. New dishes can be created and existing dishes can be deleted. We can list all the ingredients that go with a dish. We can create new ingredients and we can create new dishes."

As long as we keep on working for free, people will keep on asking us to write programs like these on a weekly basis.

.

DrawingEditor



We want to write a simple drawing program. The idea is that we can choose with radio buttons between two shapes: rectangle and circle. We also want to be able to determine whether the shapes should be filled in or not, and we want to be able to pick the color of the shapes using a combo box. In addition, we also want to be able to save and load our works of art.

We start with the UI: that's easy because in the Swing chapter we have already written a UI for such a drawing editor. As a small modification, we move the selection of the shapes to the south, and in the north region we need to place two JSButtons for loading and saving. Since we want to listen to the JSButtons, an

actionPerformed() function is required.

Try it

Since we want to save all our shapes (GObjects) later, we have to keep an eye on them. Fortunately, the JSCanvas allows us to do that:

```
let canvas = new JSCanvas();
...
let objects = canvas.getAllGObjects();
...
canvas.add(gObj);
```

When adding shapes, the user clicks on the canvas. As a consequence the canvasClicked() method is being called. But we need to know what shape is desired. So we ask the radio buttons:

```
let obj;
if (rBtnRect.isSelected()) {
    obj = new GRect(SIZE, SIZE);
} else {
    obj = new GOval(SIZE, SIZE);
}
```

Then we ask the combo box for the color,

```
const colors = [Color.RED, Color.GREEN, Color.BLUE];
obj.setColor(colors[cBoxColorPicker.getSelectedIndex()]);
```

and finally the checkbox tells us if the forms should be filled in or not:

```
if (cBtnFilled.isSelected()) {
    obj.setFilled(true);
}
```

Now we can add the shape to the canvas:

```
canvas.add(obj, _mouseX, _mouseY);
```

With this the drawing part is done.

Let's worry about the storage of our works of art. Until now, when we saved data, it was always textual data. But what about objects? For this the JavaScript Object Notation (JSON) is really helpful, and in JavaScript there is a class called JSON, that does all the complicated work.

To save our works of art, we ask the canvas to give us a list of all the GObjects. This array we then pass into the *stringify*() method of the JSON class, and the resulting JSON string, we write to file:

```
function saveFile(fileName) {
    let fw = new Utils.FileWriter(fileName);
    let objects = canvas.getAllGObjects();
    let json = JSON.stringify(objects);
    fw.write(json);
    fw.close();
    canvas.removeAll();
}
```

Conversely, reading objects is also done with the JSON class. We first read the JSON string from file:

```
function openFile(fileName) {
    canvas.removeAll();
```

```
    let fr = new Utils.FileReader(fileName);

    let json = '';
    while (true) {
        let line = fr.readLine();
        if (line == null)
            break;
        json += line;
    }

    deserialize(json);

    fr.close();
}
```

and pass it into the deserialize() function:

```
function deserialize(json) {
    const jsObj = JSON.parse(json);
    for (const el of jsObj) {

        // create raw GObject
        let gObj = undefined;
        switch (el.type) {
            case 'GOval':
                gObj = new GOval();
                break;
            case 'GRect':
                gObj = new GRect();
                break;
            default:
                print('deserialization: unknown type: ' + el.type);
                gObj = undefined;
                break;
        }

        if (gObj !== undefined) {
            // assign properties
            for (const [key, value] of Object.entries(el)) {
                // print(key + ':' + value);
                gObj[key] = value;
            }

            // add to objects and canvas
            canvas.add(gObj);
        }
    }
}
```

which then builds the GObjects again out of the JSON array. Since the JSON array is a bit stupid, we need to help it with a cast to get back our list of GObjects. Then we have to the resulting GObjects all their properties, and of course we have to add the new GObjects to our canvas. But that's all.

So what about this JSON string? Let's just take a look at it: when you open the text file "drawingEditor.json" it looks like this

```
[
    {
        "x": 147,
        "y": 96,
        "width": 20,
```

```
            "height": 20,
            "color": "red",
            "filled": false,
            "fillColor": "white",
            "type": "GRect"
        },
            ...
]
```

You clearly recognize all the information needed to build a GRect is there. Also you see that arrays are denoted with square brackets and objects are denoted with curly brackets [10]. Now I am not saying this is trivial, but it is definitely doable, and the result is quite sophisticated.

.

StockCharter

Let's get to our last and final project: StockCharter. It's about showing stock prices graphically. In a first step we write a console program that has a JSLabel, a JSTextField and a JSButton in the southern region. If the user then enters a stock symbol in the text field (e.g. "msft") and clicks on the "Graph" button, the data for the respective stock should be displayed in the console.

```
Enter stock symbol (msft): msft
StockEntry [symbol=msft,
prices=32.06,33.66,34.59,33.4157,
28.5002,27.7355,26.9466,26.2568,2
6.6413,28.2607,...]
Enter stock symbol (msft): nflx
StockEntry [symbol=nflx,
prices=246.54,222.12,223.21,209.2
75,163.47,184,181.745,96.77,85.3,
79,...]
```

Try it

1. Stock Prices

Next, we need to get the data, the stock prices. It is surprisingly difficult to get free data for stock prices, but fortunately QuantQuote has provided historical prices for the Standard & Poor's 500 (S&P 500) stock index [7]. This index includes companies such as Microsoft (msft), IBM (ibm), Ebay (ebay) and Netflix (nflx). The data can be found in the file "SP500_HistoricalStockDataMonthly.csv" and it looks like this:

```
,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,...
```

That is, it starts with the company symbol, followed by the prices, separated by commas. The first line of the file contains the respective dates.

2. Database

The next step is to read all the data from file. We do this in the StockDataBase class, which should also keep all the data, hence it is our database. We open the file "SP500_HistoricalStockDataMonthly.csv":

```
class StockDataBase {
    constructor(fileName) {
        this.stockDB = new HashMap();
        this.dates = new ArrayList();

        // open file
        let fr = new Utils.FileReader(fileName);

        // first line contains dates:
        let line = fr.readLine();
        this.readDates(line);

        // other lines contain data:
        this.readStockPrices(fr);

        // close file
        fr.close();
```

```
    }
        ...
}
```

The first line contains the dates, we process it in the method readDates() and store it in the instance variable dates. Then we read the the stock prices line by line using the method readStockPrices().

```
readStockPrices(br) {
    let line;
    while (true) {
        line = br.readLine();
        if (line == null)
            break;
        let entry = new StockEntry(line);
        this.stockDB.put(entry.getSymbol(), entry);
    }
}
```

We turn each line into a StockEntry, and store it in our HashMap, which contains all stock prices.

A StockEntry contains all data belonging to one stock, and that is symbol and prices as instance variables. The class StockEntry needs a constructor that parses a line and initializes its instance variables:

```
class StockEntry {
    constructor(line) {
        let sVals = line.split(",");
        this.symbol = sVals[0];
        this.prices = new ArrayList();
        for (let i = 1; i < sVals.length; i++) {
            if (sVals[i] == "null") {
                this.prices.add(-1.0);
            } else {
                this.prices.add(parseFloat(sVals[i]));
            }
        }
    }
    ...
}
```

It also needs a getSymbol() and a getPrices() method, and a toString() method would be nice too.

Returning briefly to the StockDataBase class: two methods are missing: findEntry() and getDates(). The first one searches for a given symbol (e.g. "msft") and should return the corresponding StockEntry from the HashMap stockDB. The second one should simply return the list with all the dates.

3. Console

To test that our database works correctly, we write a short console program:

```
async function setup() {
    createConsole();

    let db = new StockDataBase("Pr8_Stocks/SP500_HistoricalStockDataMonthly.csv");

    while (true) {
        let symbol = await readLine("Enter stock symbol (msft): ");
        if (symbol == "")
            break;

        let entry = db.findEntry(symbol);
        if (entry != null) {
```

```
            println(entry);
        } else {
            println("No entry found for: " + symbol);
        }
    }
}
```

Of course we first have to initialize our database with. This should be enough to show that our database part is working.

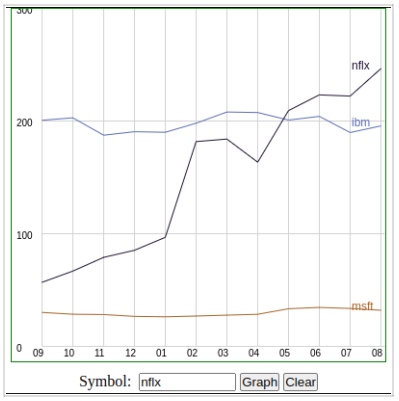
4. Graphics

Let's get to the graphics part. In the first step, we turn the console program into a UI program. Since a Program does not have a canvas for drawing, we have to take care of that ourselves. So we add the instance variable *canvas* to our program:

```
let canvas;
```

and initialize it in the setup() method:

```
canvas = new StockCanvas(db.getDates());
canvas.addStyle('width: 97%');
canvas.addStyle('height: 97%');
addWidget(canvas, 'CENTER');
```



Try it

i.e. the canvas gets a reference to the dates and we add the canvas to the CENTER area of our program. We also introduce the actionPerformed() function:

```
function actionPerformed(e) {
    if (e.getActionCommand() == "Clear") {
        canvas.clear();
    } else {
        let entry = db.findEntry(tfSymbol.getText());
        if (entry != null) {
            canvas.addEntry(entry);
        }
    }
}
```

Hence, we see three requirements for our StockCanvas class: The constructor gets a list of dates, and there must be the methods addEntry() and clear().

```
class StockCanvas extends JSCanvas {
    constructor(dates) {
        super();
        this.dates = dates;
        this.entries = new ArrayList();
        ...
    }

    clear() {
        this.entries = new ArrayList();
        this.update();
    }

    addEntry(entry) {
        this.entries.add(entry);
        this.update();
    }
```

```
            ...
    }
```

The actual drawing is done in the update() method:

```
    update() {
        this.removeAll();
        this.drawGrid();
        this.drawEntries();
    }
```

First everything from the UI is deleted, then the background grid with the captions is to be drawn, and then the stock prices. Basically, these are "just" a few GLabels and GLines in the right places.

You may not realize that now, but the StockCharter project actually stands for a whole class of similar programs: you load some data from a file or database, have some user interaction, often do some searching, and then present the results in a graphical fashion. We have come a long way.

.

QUESTIONS

1. Write a method countLinesInFile(String fileName), which returns a number (int) containing the number of lines in the file named "fileName". Use the classes BufferedReader and FileReader. You do not have to do any exception handling (try-catch).

2. If we want to read from files we usually use the FileReader. When we write our code, the compiler always forces us to use a'try-catch' block around the problematic code. Why?

3. What is this "null" thing all about?

4. Give two examples of what a HashMap is better suited for than an ArrayList.

5. A phone book is to be saved in the following HashMap:
   `let phoneBook = new HashMap();`
   Enter short code snippets that show
   a) how to insert a new phone number,
   b) how to access a person's phone number, and
   c) how to list all phone numbers.

6. Requirements: In the lecture we analyzed the requirements of the Azamon shop. Use the same technique to analyze the requirements of the Journal for Applied Computer Science:

   "The Journal has articles and issues. An article has an author, a title and a text. An issue has a title, an issue number and articles. We can list all articles, have all issues listed, as well as the articles for a specific issue number. We can create a new article and we can create new issues."

   Identify the necessary classes, instance variables, and methods. Also assign the instance variables and methods to the correct class.

7. Is there a 'setFilled()' method for the GLabel class?

.

REFERENCES

The last chapter also draws on the references of the second chapter. Also a very nice book, which is now within reach, is that of Robert Sedgewick and 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

[8] CORS on Firefox: https://dev.to/dengel29/loading-local-files-in-firefox-and-chrome-m9f

[9] Software portability, https://en.wikipedia.org/wiki/Software_portability

[10] Introducing JSON, https://www.json.org/json-en.html

.

.

JavaScript Primer

This is a very brief and superficial introduction to JavaScript. The books in the references are much better and you should get all of them. Here we only do client-side, browser-based JavaScript. Also, we use ES6. Although the class syntax is considered to be only syntactic sugar, we actually like sugar quite a lot.

JavaScript was invented by Brendan Eich in 1995. Its syntax is somewhat inspired by Java, but it has very little to do with Java. To understand some of the differences, the following comparison might be helpful:

Java	**JavaScript**
Strongly-typed	Loosely-typed
Static	Dynamic
Classical	Prototypal
Classes	Functions
Constructors	Functions
Methods	Functions

The key differences being that Java has a statically typed system, whereas JavaScript has a dynamic type system. And in JavaScript basically almost everythings seems to be a function.

With the advent of ES6 JavaScript has many new features. Most people say it is just syntactic sugar, but in my opinion it is much more than that. We will introduce a few of them here, such as classes, inheritance, closure, generators, and promises.

.

Introduction

The simplest way to run JavaScript is in the browser. Use your favorite text editor, such as Notepad, Mousepad, TextEdit, or vi, and type the following into a file you can call "index.html":

```
<html>
    <head>
        <script type="text/javascript">
            alert("Hello world!");
        </script>
    </head>
</html>
```

Now simply open it with your browser and a pop-up box saying "Hello world!" should show up. You could place the script tag also inside the body, but usually it is placed inside the head tag.

Next, let us consider *functions*, the bread and butter of JavaScript.

```
<html>
    <head>
        <script type="text/javascript">
                function run() {
                    document.write("Hello world!");
                }
        </script>
    </head>
    <body onload="run();">
    </body>
</html>
```

Here we actually use three features of JavaScript: *functions*, *events* and the *DOM*. Functions are declared inside script tags, they start with the keyword "function", often have a name and have a function body, containing JavaScript code:

```
function run() {
    document.write("Hello world!");
}
```

The word "document" refers to an HTML document and is part of the browsers DOM. DOM stands for Document Object Model and it is a JavaScript wrapper around all the different HTML elements in the browser. Most important are the *window* and the *document* objects.

JavaScript has many events, in our example we use the *onload* event:

```
<body onload="run();">
</body>
```

It is triggered when the browser has finished loading the whole HTML page. The above tells the browser to execute the JavaScript function *run()* after the HTML page has loaded completely. Interestingly, for very large pages, JavaScript may actually start running before the whole page is loaded, this can cause problems when JavaScript tries to access parts that haven't even loaded. Hence it is advisable to only start executing JavaScript code after the whole page has loaded.

Finally, let us talk about JavaScript files. Instead of mixing HTML with JavaScript code, we can also place them in separate files. In this case the "index.html" file would look like this:

```
<html>
    <head>
        <script type="text/javascript" src="script.js"></script>
    </head>
    <body onload="run();">
    </body>
</html>
```

and we put the JavaScript code inside a file called "script.js":

```
function run() {
    document.write("Hello world!");
}
```

This is clearly the preferred way of writing JavaScript code. First, it separates look (HTML) from function (JavaScript). Especially, when both get large, this is very helpful. And second, it allows for reuse, meaning, we can include the same JavaScript file in many different HTML pages.

.

Popup Boxes

For simple interaction with the user, JavaScript has three kind of popup boxes: *alert*, *confirm*, and *prompt*. The alert, we have seen already:

```
function showAlert() {
        alert("I am an alert box!");
}
```

The confirm box is often used if we want the user to verify or accept something. When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true, if the user clicks "Cancel", the box returns false.

```
function showConfirm() {
    let b=confirm("Press a button");
    if ( b === true ) {
        document.write("You pressed OK!");
```

```
    } else {
        document.write("You pressed Cancel!");
    }
}
```

The prompt box is often used if you want the user to enter some value. When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns *null*.

```
function showPrompt() {
    let name = prompt("Please enter your name", "Harry Potter");
    if ( name !== null ) {
        document.write("Hello " + name + "!");
    }
}
```

.

Variables

Let us consider some simple variable declarations:

```
x;
var y;
let z;
const c;
```

The first one declares the variable x. This maybe surprising, especially if you come from other programming languages, but JavaScript is a weakly typed language, meaning you do not have to declare variables at all. Also, you do not specify a data type. The JavaScript interpreter tries to do its best to figure this out by itself.

This seems very convenient, but may lead to problems. The most common has to do with typos: you slightly misspell a variable name. In normal languages, the compiler would give an error message that the variable was not declared. Not so in JavaScript: the interpreter thinks you want a new variable, and declares it for you automatically. However, you can tell the interpreter not to do that with "use strict", which you should.

Therefore, preferrably we use the other declarations with the keywords *var*, *let*, and *const*. The last one is always preferred, if a variable does not change, declare it a constant. The difference between the other two is subtle. Consider the following examples [2]:

```
function varTest() {
  var x = 1;
  {
    var x = 2; // same variable!
    console.log(x); // 2
  }
  console.log(x); // 2
}

function letTest() {
  let x = 1;
  {
    let x = 2; // different variable
    console.log(x); // 2
  }
  console.log(x); // 1
}
```

A variable declared with *let* is restricted to the scope in which it is defined, that is inside the curly braces.

In general, it is better to use *let*, unless one explicitely wants to create a *global* variable. Because *let*, unlike *var*,

does not allow you to create a global variable:

```
var x = "global";
let y = "global";
console.log(this.x); // "global"
console.log(this.y); // undefined
```

If you declare a variable within a function, the variable can only be accessed within that function. When you exit the function, the variable is destroyed. These variables are called local variables. You can have local variables with the same name in different functions, because each is recognized only by the function in which it is declared.

.

Data Types

In JavaScript you do not have much control over your data types. However, they do exist internally. The simplest one being

- number,
- string,
- boolean,
- and undefined.

You may notice, there is no difference between integers and floating point numbers. JavaScript infers the actual data type from the declaration:

```
let x = 42;
let y = 42.0;
let z = "42";
```

So *x* is actually treated as an integer, *y* as a floating point number, and *z* as a string. But as soon as it is needed, JavaScript will perform automatic type conversion. However, only in one direction, that is from integer to floating point, integer to string and floating point to string, but not the other way round.

For this the conversion functions are helpful:

```
let x = parseInt("42");
let y = parseFloat("42.0");
let z = Math.trunc(42.0);
```

The last one converts a floating point number to an integer. Always use *trunc()*, because *floor()* works differently on negative numbers.

Arrays and Objects

In JavaScript we also have complex data types, arrays and objects. Arrays are declared with square brackets:

```
let arr1 = [];
let arr2 = [1, 2, 3, 4];
```

The first is an empty array and the second is an array with four elements. Arrays in JavaScript can grow and shrink.

Similarily, objects are defined using curly braces, with a list of key-value pairs:

```
let cat = { name: "Garfield", age: 42 };
```

In this context also the JavaScript Object Notation (JSON) is interesting, since it is a simple way to represent these arrays and objects. For instance the above cat would become:

```
"cat": { "name": "Garfield", "age": 42 }
```

Functions

Finally, unless you are used to C's function pointers, the following may seem a little strange:

```
let f = function() { alert("hi"); };
f();
```

Here we seem to define a variable called *f*, which actually is a function, not a variable. The second line shows how *f* is being used.

typeof

You can ask the JavaScript interpreter what the type of a given variable is with the *typeof* operator. For instance, the following code

```
let x = 42;
console.log( typeof(x) );
```

would output "number" to the console. You may want to try the following examples to see how the typeof operator works.

```
function run() {
    let u1;
    console.log(typeof (u1));  // undefined
    let u2 = undefined;
    console.log(typeof (u2));  // undefined
    let x = 42;
    console.log(typeof (x));   // number
    let y = 42.0;
    console.log(typeof (y));   // number
    let z = "42";
    console.log(typeof (z));   // string
    let b = true;
    console.log(typeof (b));   // boolean
    let n = null;
    console.log(typeof (n));   // object
    let arr = [];
    console.log(typeof (arr)); // object
    let obj = { name: "Garfield", age: 42 };
    console.log(typeof (obj)); // object
    let f = function () { };
    console.log(typeof (f));   // function
}
```

Interesting maybe that *null*, *arrays* and *objects* are all considered objects.

.

Operators

JavaScript has the standard arithmetic and assignment operators: =, +, -, *, /, %, ++ and --. The + operator can also be used to concatenate strings. When adding a number and a string, the result will be a string.

Comparison and logical operators are used to test for true or false. We have the standard comparison operators

```
==      equal
!=      not equal
>       larger than
<       less than
>=      larger than or equal
<=      less than or equal
```

In addition, JavaScript has two more comparison operators: the exactly equals === and the not exactly equals !==. They compare not only value but also type. The following example shows the difference:

```
let x = 42;
let y = "42";
if (x == y) {
    console.log("x and y are equal");
}
if (x === y) {
    console.log("x and y are exactly equal");
}
```

In general, always use the exactly equals operators.

As for the logical operators again those work exactly as in Java or C++.

As for the bitwise operators, strange things might happen here, and they are slow. So no point in using them.

.

Conditional Statements

As for conditions, JavaScript has the standard if-else statement:

```
let dt = new Date()
let time = dt.getHours()
if (time < 10) {
    alert("Good morning");
} else if (time > 10 && time < 16) {
    alert("Good day");
} else {
    alert("Good evening");
}
```

In addition, there is also the standard *switch* statement which works as expected.

.

Loops

JavaScript has the standard for-loop:

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```

Important here is the *let*, because if you forget it, JavaScript won't complain, but it will create a global variable *i* for you, which is most likely not what you intended. There is also the while-loop and the do-loop.

However, we have two more loops in JavaScript: the *for...in* loop and the *for...of* loop.

for...in Loop

The for...in loop is used in connection with objects, when you want to iterate through all the properties of an object.

```
let cat = { name: "Garfield", age: 42 };
for (const property in cat) {
    console.log(property + ": " + cat[property]);
}
```

for...of Loop

The for...of loop is used when you want to iterate over an iterable object. Iterable objects are Arrays, Strings, TypedArrays, Maps, and Sets, for instance.

```
let arr = [1, 2, 3, 4];
for (const element of arr) {
    console.log(element);
}
```

JavaScript also has the *break* and *continue* statement, but if possible those should be avoided.

.

Functions

As we have seen, functions are at the core of JavaScript. In JavaScript almost everything is a function. Functions can be called by events or by other functions. Functions can have parameters and they can be anonymous. There can also be functions inside functions.

JavaScript distinuishes between function declaration and function expression.

Function Declaration

This is most likely the way you have declared functions before. Here the function has a name and parameters, they can also return something, and they exist on the global scope.

```
function run() {
    print("hi there");
}

function print(msg) {
    console.log(msg);
}
```

Function Expression

Function expressions on the other hand look like they are assigned to a variable. A function expression is only available after it has been declared, and it is not neccessarily of global scope.

```
function run() {
    const print = function (msg) {
        console.log(msg);
    }
    print("hi there");
}
```

Notice the space between the keyword *function* and the parameter list *()*. This is convention.

Function expressions are also called anonymous functions. To see why, first notice that in JavaScript functions can also be passed as parameters to other functions. We declare two functions, *createMsg()* and *print()*, and we pass the result of one into the other:

```
function run() {
    const print = function (msg) {
```

```
        console.log( msg );
    }

    const createMsg = function () {
        return "hi there";
    }

    print(createMsg());
}
```

We can rewrite this with an anonymous function:

```
function run() {
    const print = function (f) {
        console.log( f() );
    }

    print(function () {
        return "hi there";
    });
}
```

You may notice that the *createMsg()* function has disappeared, the code became shorter, but not neccessarily easier to read.

Parameters

In other languages you may have seen something called *overloading*, where you have two functions with the same name, but different types or number of parameters. Since JavaScript is weakly typed that does not make a lot of sense, and therefore is not possible. JavaScript's solution to the problem is as simple as it is ingenious: simply define the function with the maximum number of parameters, but when calling it, you do not have to give all parameters.

```
function run() {
    print("hi there", "how are you?");
    print("hi there");
}

function print(msg1, msg2) {
    console.log(msg1);
    console.log(msg2);
}
```

The first version behaves as expected, whereas in the second version, the parameter *msg2* is *undefined*. However, the code just runs fine. If you do not like things to be *undefined*, which is perfectly understandable, you can set parameters to a predefined value:

```
function print(msg1, msg2 = "empty") {
    console.log(msg1);
    console.log(msg2);
}
```

.

Error Handling

Because JavaScript is a weakly typed, interpreted language, it is very easy to use, but it is also very easy to make mistakes. Fortunately, it has a pretty elaborate error handling mechanism, using the try catch syntax.

try catch

For instance, assigning a variable to something that does not exist, will cause a *ReferenceError* (assuming you use "use strict"):

```
function run() {
    let x = y;
}
```

If you want to catch this error, you would surround it with a try catch block, The difference is: without our program will crash, whereas with it will print an error message, and continue to run:

```
function run() {
    try {
        let x = y;
    } catch (error) {
        console.log("An error occured: " + error);
    }
}
```

The error object contains information about what kind of error occured, and we can decide what to do next depending on the error.

In JavaScript, there are many errors, such as *TypeError*, *RangeError*, *ReferenceError*, and *SyntaxError*:

```
function run() {
    // TypeError
    try {
        let fritz;
        fritz.setColor(Color.RED);

        let num = 1;
        num.toUpperCase();

    } catch (e) {
        console.log(e);
    }

    // RangeError
    try {
        let b = new Array(-1)

    } catch (e) {
        console.log(e);
    }

    // ReferenceError
    try {
        let x = 3;
        x = x + y;

    } catch (e) {
        console.log(e);
    }

    // SyntaxError
    try {
        eval("let s = 'hi");

    } catch (e) {
        console.log(e);
```

```
    }
    console.log("still running...");
}
```

Interesting maybe that what would be errors in other languages, are no problem for JavaScript:

```
function run() {
    try {
        // ArithmeticException
        let x = 5 / 0;
        console.log(x);        // Infinity

        // NumberFormatException
        let y = parseInt("five");
        console.log(y);        // NaN

        // ArrayIndexOutOfBoundsException
        let eggs = [0, 1, 2, 3];
        console.log(eggs[5]);  // undefined

    } catch (e) {
        console.log("No errors here");
    }
    console.log("still running...");
}
```

throw

If you want to, you can also throw errors. For instance,

```
if (!locateKarel()) {
    throw Error("Could not locate Karel in the world");
}
```

would throw an error, which then should be caught with some try catch.

"use strict"

While we are at errors: recall that in JavaScript we are not required to declare variables. This can lead to problems:

```
//"use strict";  // uncomment this line

function run(() {
    let fritz = 5;
    frits = 4;
    console.log(fritz);
}
```

In the third line, we want to reassign a new value to the variable *fritz*. But we made a typo, and hence instead of a reassignment, we declared a new variable named *frits*. JavaScript does not care, there is no error here. To avoid these kinds of stupid mistakes, it is highly recommended that you start all of your JavaScript code with the "use strict" keyword. Adding it, will result in an error being thrown.

.

DOM

JavaScript per se is independent of the Document Object Model (DOM). But as soon as we say things like

```
document.write("Hello World");
```

```
console.log("hi there");
```

we use the DOM. Both *document* and *console* are part of the browser, one being the HTML document displayed, the other the browser's console. For JavaScript they are just objects with certain properties and methods. So DOM is just an abstraction of the browser, and whenever we want to interact with the browser, we do that via the DOM.

The most important objects the DOM provides are the *console*, which is mostly used for debugging, the *document*, which allows use to modify the HTML shown, and the *window* object, which gives us information about the browser window, for instance its size. As an example for what the window object might be good for consider the following code:

```
function main() {
    alert("Done loading!");
}
window.onload = main();
```

The last line tells the browser to call the main() function, once the page has completely loaded.

To see why the *document* object is useful consider the following two examples. Assume we have the following HTML document:

```
<html>
    <head>
        <script type="text/javascript" src="script.js"></script>
    </head>
    <body onload="run();">
        <textarea id="hans">Hi there</textarea>
    </body>
</html>
```

where we have a text area element, which has the id named "hans". We can access this element with the *getElementById()* method:

```
function run() {
    const element = document.getElementById("hans");
    console.log(element.innerHTML);
}
```

We can also add, modify, and remove HTML elements. Assume we have an empty HTML document and we want to add a text area. The following code would just do that:

```
function run() {
    // create text area
    let textarea = document.createElement('textarea');
    textarea.id = 'hans';
    textarea.value = 'Hi there';
    textarea.rows = 10;
    textarea.cols = 32;
    textarea.readOnly = true;

    // add text area to body
    const _body = document.getElementsByTagName('body')[0];
    _body.appendChild(textarea);

    textarea.focus();
}
```

This should give you an impression of the power of the DOM.

.

Events

Interaction with the user usually happens via events. But there are also events coming from the browser, such as the *onload* event, we have already seen. Examples of events are

- loading of a web page or an image,
- a mouse click or a keystroke,
- submitting an HTML form,
- or a timer event.

We need to tell the browser which events we are interested in, and what to do when the event occurs. This can be done in the HTML or via JavaScript.

An HTML example we have seen already:

```
<html>
    <head>
        <script type="text/javascript" src="script.js"></script>
    </head>
    <body onload="run();">
    </body>
</html>
```

Here we tell the browser that if the *onload* event occurs, that it should call the *run()* function. We can do the same thing with JavaScript:

```
function run() {
    document.write("Hello world!");
}
window.onload = run();
```

If you are interested in key events, you would add an event listener:

```
function run() {
    document.addEventListener('keydown',
        function (ev) {
            console.log(ev.key + "," + ev.code);
        }
    );
}
```

In the same way you can add all kinds of events, such as *keydown*, *keypress*, *click*, *onmousedown*, *change* and many more.

Two interesting functions are the *setTimeout()* and *setInterval()* functions: the first allows you to wait a given time before a function is being called, and the second one calls a function repeatedly.

```
function run() {
    setInterval(sayHi, 1000);
}

function sayHi() {
  document.write("hi there<br/>");
}
```

Please, notice that inside the *setInterval()* function we are not calling "sayHi()", but instead "sayHi". This makes a big difference, try it for yourself, and make sure you understand the difference!

.

.

eval()

Some people say eval() is evil, I personally think it is the greatest thing since sliced bread. Sure it is dangerous, especially in a web language, but which other language lets you dynamically execute arbitrary new code during run-time?

```
function run() {
    let code = prompt("Enter some JavaScript code:", "document.write('hi');");
    eval(code);
}
```

If this does not shock you, nothing will [6].

.

Classes

One of the most welcome additions to JavaScript are classes. The syntax looks quite similar to Java's, however, there are subtle differences:

```
function run() {
    const hansel = new Student("Hänschen");
    console.log(hansel.name);
    console.log(hansel.toString());
}

class Student {
    constructor(nm) {
        this.name = nm;
    }

    toString() {
        return "Student [name=" + this.name + "]";
    }
}
```

You will immediately notice that the constructor is actually called *constructor*. Creation of objects using the *new* keyword also may look familiar.

Interesting should be the line

```
this.name = nm;
```

which seems to indicate that there is an instance variable called *name*, but if we look, we do not find a declaration. In fact, there is none, this is just JavaScripts way of defining instance variables, very efficient I would say.

Another interesting thing, there is no overloading in JavaScript, neither for the constructor nor for methods, we talked about this when we were talking about functions before.

Closures

Let's return to the instance variables. All instance variables defined above are public, meaning anybody can see and modify them. This violates the principle of information hiding. But not all is lost, because we have *closures* in JavaScript. Assume our Student class should also have a read-only, private property *id*. This is how we would implement this in JavaScript:

```
class Student {
    constructor(name, id = -1) {
        this.name = name;
```

```
            // closure: read only
            this.getId = (function () {
                let _id = id;
                return function () {
                    return _id
                }
            })();
        }

        toString() {
            return "Student [name=" + this.name + ", id=" + this.getId() + "]";
        }
    }
```

While we are at it, we can also use closures to implement an increment-only method:

```
    class Student {
        constructor(name, id = -1) {
            this.name = name;

            ...

            // closure: increment only
            this.incrementCredits = (function (x) {
                let _credits = 0;
                return function (x) {
                    if (x !== undefined && x > 0) {
                        _credits += x;
                    }
                    return _credits
                }
            })();
        }

        toString() {
            return "Student [name=" + this.name + ", id=" + this.getId()
          + ", credits=" + this.incrementCredits() + "]";
        }
    }
```

If we call *incrementCredits()* without arguments, it just returns the current value of credits, if we call it with an argument, it will increment the credits by that amount.

Inheritance

We use the keyword *extends* to inherit from a parent class:

```
    class Freshman extends Student {
            constructor(name, id = -1) {
                    super(name, id);
            }

        toString() {
            return "Freshman [name=" + this.name + ", id=" + this.getId()
          + ", credits=" + this.incrementCredits() + "]";
        }
    }
```

If you know Java this may look familiar to you. As you can see we can override methods if we desire, but we don't have to. And there is the *super* keyword to call the parent's class constructor or methods of the parent class.

Inheritance Chain

How about methods of the grandparent class, can we call those as well? JavaScript has a method named *getPrototypeOf()* that allows you to get a handle to the respective parent, kind of neat:

```
function run() {
    const hansel = new Freshman("Hänschen", 12345);

    let me = Object.getPrototypeOf(hansel);
    console.log(me.constructor.name);           // Freshman
    let parent = Object.getPrototypeOf(me);
    console.log(parent.constructor.name);       // Student
    let grandParent = Object.getPrototypeOf(parent);
    console.log(grandParent.constructor.name);  // Object

    parent.test();
}
```

.

Polymorphism

If you lookup the meaning of polymorph, it says something like "an object or material which takes various forms". We stay with our Student and Freshman classes. If we have two students, *hansel* and *gretel*, *hansel* being a Freshman and *gretel* being a Student, then actually both of them are Students:

```
function run() {
    const hansel = new Freshman("Hänschen");
    polymorphic(hansel);
    const gretel = new Student("Gretchen");
    polymorphic(gretel);
    polymorphic("hi there");
}

function polymorphic(studnt) {
    console.log(studnt.toString());
}
```

Now in our *polymorphic()* method, we assume that the parameters passed in are Students, and therefore have the *toString()* method.

So using polymorphism in JavaScript is straight forward, the problem is enforcing it: because JavaScript is weakly typed, we can pass actually anything we want. However, again not all is lost, because we can enforce proper datatypes using the *instanceof* operator:

```
function polymorphic(studnt) {
    if (studnt instanceof Student) {
        console.log(studnt.toString());
    } else {
        throw Error("Parameter is not of type Student!");
    }
}
```

.

Static Methods

JavaScript also has *static* methods, an example is the Random class:

```
class Random {
    static nextInt(low, high) {
```

```
            if (high !== undefined) {
                return parseInt(low + Math.random() * (high - low));
            } else {
                return this.nextInt(0, low);
            }
        }

        static nextDouble(low, high) {
            if (high !== undefined) {
                return low + Math.random() * (high - low);
            } else {
                return this.nextDouble(0, low);
            }
        }
    }
```

We call these methods directly, i.e., no object needs to be instantiated:

```
    function run() {
        console.log( Random.nextInt(2,7) );
    }
```

Another example inspired from Java is *System.currentTimeMillis()*:

```
    class System {
        static currentTimeMillis() {
            return new Date().getTime();
        }
    }
    // const time = System.currentTimeMillis();
```

Don't use static to often, it does more harm than good.

.

Enums and Constants

A neat trick on how to create "constants" or something resembling enums is in the following way:

```
    class Color {
    }
    Color.RED = 'red';
    Color.GREEN = 'green';
    Color.BLUE = 'blue';
    Color.WHITE = 'white';
    Color.BLACK = 'black';
```

Although neat, be careful, they are not constants!

.

Namespaces

Another thing kind of missing in JavaScript are *namespaces*. But as usual there is a trick. We define an empty object called *Utils* as a global variable, and then attach our class to it as property:

```
    var Utils = {}
    Utils.Random = class {
        static nextInt(low, high) {
            if (high !== undefined) {
```

```
            return parseInt(low + Math.random() * (high - low));
        } else {
            return this.nextInt(0, low);
        }
    }
}
```

When using it, it looks exactly like a namespace or package would:

```
function run() {
    console.log( Utils.Random.nextInt(2,7) );
}
```

But naturally, it is not.

.

Promise

Whenever you have to wait for something, most likely you will need a promise. Let's look at two simple examples.

Assume you want something to happen once per second, that is you want a *pause()* method. Then the following is an implementation using a Promise:

```
const pause = function (milliseconds = 500) {
    return new Promise(
        function (resolve) {
            return setTimeout(resolve, milliseconds);
        })
};
```

You would call this method in the following way:

```
async function run() {
    let i = 0;
    while (true) {
        console.log("waiting one second: " + i++);
        await pause(1000);
    }
}
```

Notice the *async* and *await* keywords: *async* tells us that the function run() is asynchronous, and the *await* tells us that at this point execution might stop until something happens.

Another nice example is asking the user for feedback [3]:

```
function promptForDishChoice() {
    return new Promise(function (resolve, reject) {
        const dialog = document.createElement("dialog");
        dialog.innerHTML = '<form method="dialog">' +
            '<select><option value="pizza">Pizza</option>' +
            '<option value="pasta">Pasta</option></select>' +
            '<button value="cancel">Cancel</button>' +
            '<button type="submit" value="ok">OK</button></form>'

        dialog.addEventListener("close", function () {
            if (dialog.returnValue === "ok") {
                return resolve(dialog.querySelector("select").value);
            } else {
                return reject(new Error("User cancelled dialog"));
            }
```

```
        });

        document.body.appendChild(dialog);
        dialog.showModal();
    });
}

async function run() {
    const choice = await promptForDishChoice();
    console.log('choice=' + choice);
}
```

To understand this, first look at the *addEventListener()* part: if the user clicked "ok" we return a *resolve()*, meaning the promise was resolved. Otherwise, we return a *reject()*, meaning the promise was rejected.

This shows very nicely the idea behind promises: a promise is in one of three states:

- pending: this is the initial state, it is neither resolved nor rejected,
- resolved: meaning that the operation was completed successfully, or
- rejected: meaning that the operation has failed.

.

Generators

JavaScript is a synchronous language, meaning there is a single thread. Events are an exception, so are timer and asynchronous web requests. So obviously asynchronous programming seems possible in JavaScript. A callback is also an example of asynchronous programming. And generators are another.

A generator function is defined in the following way:

```
function* generator(i) {
    yield 1;
    yield 2;
    yield 3;
}
```

We notice that instead of *returns* it has *yields*. We create a function expression *generate* and then call the *next()* method:

```
function run() {
    const generate = generator();
    console.log(generate.next().value);  // 1
    console.log(generate.next().value);  // 2
    console.log(generate.next().value);  // 3
    console.log(generate.next().value);  // undefined
}
```

We observe that after every call to *next()* we go from one yield to the next yield. So in a sense *yield* is similar to *return*, but it keeps state, meaning it remembers where it left of last time, and continues from there when it is called again. This may seem useless, but let's consider two more examples.

Assume we want a function that generates the sequence 0, 1, 2, 3, 3, 3, ... then we could do this with the following generator function:

```
function* generator(i) {
    for (let i = 0; i < 3; i++) {
        yield i;
    }
    return 3;
}
```

Or if we want a function that is called exactly four times, once per second? We use the same generator function as above, but our *run()* method looks like this:

```
async function run() {
    const generate = generator();
    while (true) {
        let result = generate.next();
        console.log("next: ", result);
        await pause(1000);
        if (result.done) {
            break;
        }
    }
}
```

where we used the *pause()* method from above. Notice that *result.done* is true when we hit the return statement in the generator [5].

.

Arrow Function

Another new feature of ES6 is the arrow function. You will not see it much in my code, because for beginners it may be a bit confusing. Consider the following code with arrow function:

```
const pause = (milliseconds = 500) =>
    new Promise(resolve => setTimeout(resolve, milliseconds));
```

Very concise, but maybe not so easy to understand on first sight. The same code without arrow function:

```
const pause = function (milliseconds = 500) {
    return new Promise(
        function (resolve) {
            return setTimeout(resolve, milliseconds);
        })
};
```

It definitely is more code to write, but now it becomes clear that what you are doing here is declaring two anonymous functions [4].

.

REFERENCES

[1] JavaScript: The Good Parts: Working with the Shallow Grain of JavaScript, Douglas Crockford

[2] let, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

[3] Promise, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[4] When should I use arrow functions in ECMAScript 6?, https://stackoverflow.com/questions/22939130/when-should-i-use-arrow-functions-in-ecmascript-6

[5] Generator: Using the function* Declaration in JavaScript, https://www.geeksforgeeks.org/using-the-function-declaration-in-javascript/?ref=rp

[6] eval(), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

.

.

LIBRARIES

How does it all work? In this chapter we take a look behind the scenes. If you look at the tryItFrame.html file, you will notice the following imports in the head section:

```
<head>
  ...
  <!-- order is importent! -->
  <script src="./libraries/p5.min.js"></script>
  <!-- <script src="./libraries/p5.sound.min.js"></script> -->
  <script src="./libraries/gui.js"></script>
  <script src="./libraries/graphics.js"></script>
  <script src="./libraries/console.js"></script>
  <script src="./libraries/utils.js"></script>
</head>
```

The first library is the standard p5.js library from Processing. The other four are what we will explain on the pages below.

.

Console Programs

A console program is an HTML text area widget that is used to display text. Console programs are very simple, and you can use the following commands

- print()
- println()
- readLine()
- readInt()
- readDouble()
- clear()

Using a text area widget for a console seems straight forward, but there are a few things that makes this tricky:

1. the functions print() and clear() are already defined in p5.js
2. there is no read() function, the only thing that allows user input is the prompt() function

Example

A simple example of our console program looks like this:

```
async function setup() {
  createConsole();

  println('This program adds two numbers.');
  let n1 = await readInt('Enter number one: ');
  let n2 = await readInt('Enter number two: ');
  let sum = n1 + n2;
  println( 'The sum is: ' + sum );
}
```

You may notice the somewhat unusual *async* and *await* keywords. More on this in a moment.

print() and clear()

Since both print() and clear() already exist, we need to redefine them. This is done in the createConsole() function:

```
function createConsole(_ROWS, _COLS) {
```

```
    // redefine print():
    print = function (msg) {
        if (msg === undefined) {
            msg = '';
        }
        // textarea.append(msg);
        textarea.value += msg;
    }

    // redefine clear():
    clear = function () {
        textarea.value = '';
    }
}
```

readLine()

The hard part is reading from console. JavaScript only has the prompt to do that. And JavaScript does not have a pause(). One elegant way to get the desired functionality is via promises.

```
function readLine(msg) {
    if (msg === undefined) {
        msg = '';
    }
    textarea.value += msg;

    enterPressed = false;
    textEntered = '';
    return new Promise((resolveOuter) => {
        resolveOuter(
            new Promise((resolveInner) => {
                var check = function () {
                    if (enterPressed) {
                        // print('Enter');
                        textarea.value += '\n';
                        resolveInner(textEntered);
                    } else {
                        // print('No Enter');
                        setTimeout(check, 100); // check again in a second
                    }
                }
                check();
            })
        );
    });
}
```

This is the reason why we have to use async and await when using any of the read function.

.

Graphics Programs

A graphics program uses the HTML canvas tag for drawing and animation. However, we do not use it directly, but instead use p5.js for most of the drawing. Our graphics library is basically a wrapper around p5.js. In our graphics programs you can use the following commands:

- add(obj, x, y)
- getElementAt(x, y)
- getElementsAt(x, y)
- removeAll()

- removeObj(obj)
- setBackground(color)
- update()

p5.js

p5.js comes from the Processing Foundation, that also created Processing for Java. The idea is very similar, make JavaScript as easy to use as possible. We use p5.js, because it is well developed and maintained, and it runs on most browsers, even the mobile ones. Also, a key feature is its support and abstraction for various hardware, such as webcam and microphone, but also playing videos and 3D drawing are made easy with p5.js.

Basically all p5.js programs have the same structure:

```
function setup() {
    createCanvas(400, 400);
    frameRate(25);
    //noLoop();
}

function draw() {
    background(220);
        ellipse(0, 0, 50, 50);
}
```

The *setup()* method is called once at the start of the program. In addition there is a *draw()* method that is called repeatedly, depending on what frame rate was chosen. It is also possible to not call the *draw()* method at all via the *noLoop()* command. The *ellipse()* function is one of p5.js many drawing functions.

If you save the above code in a file named *sketch.js*, then the following HTML is all that is needed to run it:

```
<html>
  <head>
    <script src="https://cdn.jsdelivr.net/npm/p5@1.6.0/lib/p5.js"></script>
    <script src="sketch.js"></script>
  </head>
  <body>
  </body>
</html>
```

Naturally, all the magic lies in the p5.js file.

Graphics Objects

Writing code in p5.js is not all that object oriented, or rather, there is a strong tendency to not write object oriented code when working with p5.js. For small projects that is perfectly allright. But as soon as things get a little more complicated and objects start interacting, like collisions, or a user clicking on them, things get rather tedious.

This is why we wrote wrapper graphics objects around the p5.js drawing functions, strongly inspired by the ACM graphics library for Java:

- GArc
- GCompound
- GImage
- GLabel
- GLine
- GObject
- GOval
- GPixel
- GPolygon
- GRect
- GVideo
- GVideoPreview

- GWebCam
- GWebCamPreview

Most graphics objects are quite simple, GImage and GCompound maybe a little more involved.

Example

Just to get an idea how the combination of p5.js and graphics objects work, consider the following example:

```
function setup() {
  createCanvas(300, 150);
  frameRate(5);

  let fritz = new GRect(150, 100, 50, 50);
  fritz.setColor(color('blue'));
  fritz.setFilled(true);
  fritz.setFillColor(color('yellow'));
  add(fritz);
}

function draw() {
  update();
}
```

In the setup we create the canvas, set the frame rate, and then create a GRect object. In the *draw()* function we simply call the *update()* function. The *update()* function is very simple:

```
function update() {
    background(backgroundColor);
    for (let i = 0; i < gobjects.length; i++) {
        gobjects[i].draw();
    }
}
```

It iterates through all the gobjects and calls their respective *draw()* functions. The *add()* method above, adds graphics objects to the *gobjects* array.

```
function add(obj, x, y) {
    if ((x !== undefined) && (y !== undefined)) {
        obj.setLocation(x, y);
    }
    gobjects.push(obj);
}
```

GObject

The class GObject is the parent class of all other graphics objects:

```
class GObject {

    constructor(x, y, width, height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.color = 'black';
        this.filled = false;
        this.fillColor = 'white';
        ...
    }
```

```
    contains(x, y) {
        ...
    }
    ...
}
```

Meaning every graphics object has a location and a size, a color, a fillColor and is either filled or not. GObject should be considered an abstract class, but JavaScript does not have that concept.

As a concrete implementation of GObject we will only consider GRect, all other graphics classes are very similar:

```
class GRect extends GObject {

    constructor(x, y, width, height) {
        if ((width === undefined) && (height === undefined)) {
                        // new GRect(50,50) has width and height = 50, but x and y = 0!
            super(0, 0, x, y);
        } else {
            super(x, y, width, height);
        }
    }

    draw() {
        noFill();
        stroke(this.color);
        rect(this.x, this.y, this.width, this.height);
    }
}
```

Here the functions *noFill()*, *stroke()*, and *rect()* are standard p5.js functions.

GImage

Of the graphics classes the GImage class is a little more complicated, because loading of images in p5.js happens asynchronously. p5.js has the *loadImage(path, [successCallback], [failureCallback])* function, which allows for a callback in case of success or failure. Per se, this is not a problem. However, if we want to access the pixels of the image via the *getPixelArray()* method, there could be a problem: assume it takes a little longer for the image to load, then the call to the *getPixelArray()* method might happen before the image is loaded.

To solve this, we need two steps: First, we introduce a property *imageLoaded*, which initially is set to false, and only after the image has been successfully loaded it will be true. But here comes another subtle problem: the *this* of the callback is different from the *this* of our GImage. The standard solution is to get a value on the GImage's *this* and call it *that*:

```
...
this.pixelArray1 = [];
this.imageLoaded = false;
let that = this;
this.image = loadImage(imageName, img => {
    image(that.image, that.x, that.y);
    that.image.loadPixels();
    // remember pixels for modification
    for (let i = 0; i < that.image.pixels.length; i++) {
        that.pixelArray1[i] = that.image.pixels[i];
    }
    that.image.updatePixels();
    that.imageLoaded = true;
});
...
```

Second, before allowing usage of the *getPixelArray()* method, we need to make sure, that the image has loaded, i.e.

we have to wait until *imageLoaded* is true. We can do that using a promise:

```
getPixelArray() {
    return new Promise((resolveOuter) => {
        let that = this;
        resolveOuter(
            new Promise((resolveInner) => {
                var check = function () {
                    if (that.imageLoaded) {
                        resolveInner(that.pixelArray1);
                    } else {
                        setTimeout(check, 100); // check again in a second
                    }
                }
                check();
            })
        );
    });
}
```

Subtle, also here we have to remember the *this* using *that*. As I said, it is a little tricky.

.

UI Programs

For writing UI programs, we use standard HTML widgets with thin JavaScript wrappers around them. A design challenge was: could you make the resulting code look almost like a Java Swing program? Turns out that very little has to be changed, for the standard HTML widgets to behave like their corresponding Swing counterparts.

In UI programs you can use the following commands:

- createGUI(width, height)
- addWidget(obj, where)
- removeWidget(obj, where)
- setLayout(_layout, _cols)
- addActionListener(obj)
- addChangeListener(obj)

Example

To see how a simple UI program is implemented with our UI library, consider the following example:

```
let tf;

function setup() {
    createGUI(300, 150);

    setLayout('border');

    let lbl = new JSLabel("Name: ");
    addWidget(lbl, 'SOUTH');

    tf = new JSTextField(10);
    addWidget(tf, 'SOUTH');

    let btn = new JSButton("Login");
    addWidget(btn, 'SOUTH');
}

function actionPerformed(ev) {
    print("Name: " + tf.getText());
```

```
}
```

If you are familiar with Swing, you will recognize the respective classes easily.

UI Widgets

Just for demonstration purposes we implemented a handful of the UI widgets:

- JSObject
- JSLabel
- JSTextField
- JSTextArea
- JSAbstractButton
- JSButton
- JSCheckBox
- JSRadioButton
- JSComboBox
- JSLink
- JSFileUpload
- JSOptionPane
- JSPanel
- JSCanvas

If desired you can add as many as you like. Most are really easy to implement, but somewhat trickier were the JSPanel and the JSCanvas.

JSObject

The class JSObject is the parent class of all other UI widgets:

```
class JSObject {
    constructor() {
        this.element = document.createElement('span');
    }

    setStyle(css) {
        this.element.style = css;
    }

    addStyle(css) {
        this.element.style.cssText += css;
    }

    getWidth() {
        return this.element.width;
    }

    getHeight() {
        return this.element.height;
    }
}
```

Every widget is based on an underlying HTML element, in case of the JSObject it is the tag. Look and feel is changed via cascading style sheets. Being able to use CSS for styling is very convenient. Really, there is not much more to this.

As a second example, let's look at the JLabel:

```
class JSLabel extends JSObject {
    constructor(text) {
        super();
        this.element = document.createElement('span');
```

```
            this.element.style.padding = DEFAULT_PADDING;
            this.element.style.margin = DEFAULT_MARGIN;
            this.element.innerHTML = text;
        }

        getText() {
            return this.element.innerHTML;
        }

        setText(text) {
            this.element.innerHTML = text;
        }
    }
```

I think you get the idea. A side note: if you have read Douglas Crockford's book, then you know that using *innerHTML* is not such a good idea. If you have not read Douglas Crockford's book, you should.

Event Handling

Again, we don't really have to do much. The standard HTML events will do just fine. Whenever we add a widget, we add an EventListener for it:

```
    function addActionListener(obj) {
        if (obj instanceof JSAbstractButton) {
            obj.element.addEventListener('click',
                function (ev) {
                    let ae = new ActionEvent(ev.target.id, obj);
                    window.actionPerformed(ae);
                }
            );
        } else if (obj instanceof JSCanvas) {
            obj.element.addEventListener('click',
                function (ev) {
                    const rect = obj.element.getBoundingClientRect();
                    window.canvasClicked(ev.clientX - rect.left,
                                         ev.clientY - rect.top);
                }
            );
        } else if (obj instanceof JSTextField) {
            obj.element.addEventListener("keyup",
                function (ev) {
                    if (ev.key === "Enter") {
                        let ae = new ActionEvent(ev.target.id, obj);
                        window.actionPerformed(ae);
                    }
                }
            );
        }
    }
```

Adding an EventListener for every widget seems a little overkill, but we do not expect very large programs to be written with this library.

Just in case you are curious, the ActionEvent is a really thin wrapper around two instance variables, *command* and *source*:

```
    class ActionEvent {
        constructor(id, obj) {
            this.command = '' + id;
            this.source = obj;
        }
```

```
    getActionCommand() {
        return this.command;
    }

    getSource() {
        return this.source;
    }
}
```

JSPanel

Now the one class that is a really tricky is the JSPanel class. From an HTML point of view it is just a tag. What makes it complicated are two things: first, it can have child widgets, including JSPanels. And second, it can have three different layouts: *flow*, *grid* and *border*.

Layout

Let's talk about the layouts first: *flow* and *grid* are simple, all we do we set the style of the tag. In the case for *flow* it is one line,

```
this.element.style.display = 'inline-block';
```

in the case for *grid* it is four lines:

```
this.element.style.display = 'inline-block';
this.element.setAttribute('class', 'wrapper');
this.element.style.display = 'grid';
this.element.style.gridTemplateColumns =
        'repeat(' + _cols + ', ' + Math.trunc(100 / _cols) + '%)';
```

If you know your CSS the above should make some sense.

The tricky one is the *border* layout. For this we use the *flex* layout. This can be horizontal or vertical. So we are not doing a real border layout, but more of a box layout. But if you combine a horizontal box layout and nest inside it a vertical box layout, you do have a border layout.

So the first thing we do, we dynamically add a style to an existing HTML page, by inserting it into the <head> tag:

```
let stle = document.createElement("style");
stle.innerHTML = ".flexbox-parent {
        width: 100%; height: 100%; display: flex; flex-direction: column;
        justify-content: flex-start; align-items: stretch; align-content: stretch;}";
stle.innerHTML += ".flexbox-item-grow { flex: 1;}";
stle.innerHTML += ".fill-area-content { overflow: auto;}";
document.getElementsByTagName("head")[0].appendChild(stle);
```

Now, this is totally not needed, and I am sure will cause real problems if you had several of them, but I just thought this is so cool...

Anyways, once we have the style, we use it:

```
this.element.style.display = '';
this.element.setAttribute('class', 'flexbox-parent');
if (_cols == 'horizontal') {
    this.element.style.flexDirection = 'row';
}

this.north = new JSPanel('flow');
this.north.element.setAttribute('class', 'flexbox-item header');
this.north.element.setAttribute('align', 'center');
this.element.appendChild(this.north.element);
```

```
this.center = new JSPanel('flow');
this.center.element.setAttribute('class', 'fill-area-content flexbox-item-grow');
this.center.element.setAttribute('align', 'center');
this.element.appendChild(this.center.element);

this.south = new JSPanel('flow');
this.south.element.setAttribute('class', 'flexbox-item footer');
this.south.element.setAttribute('align', 'center');
this.element.appendChild(this.south.element);
```

Again, if you know your CSS, this is pretty straight forward. But you also see, why this is a little complicated, we introduce three new JSPanels, one for each region, *north*, *center*, and *south*.

Child Widgets

Adding the children is trivial for *flow*, and *grid*: you simply call *this.element.appendChild()*. For the *border* layout, we need to make sure we add it to the correct child:

```
add(obj, where) {
    ...

    if (this.layout == 'border') {
        if (where.toLowerCase() == 'north' ||
                        where.toLowerCase() == 'west') {
            this.north.add(obj);
        } else if (where.toLowerCase() == 'south' ||
                                        where.toLowerCase() == 'east') {
            this.south.add(obj);
        } else {
            this.center.add(obj);
        }

    } else if (this.layout == 'grid') {
        this.element.appendChild(obj.element);

    } else {
        this.element.appendChild(obj.element);
    }
}
```

What we did not show, is code that adds the event listeners and treats the case where we add a JSPanel to a JSPanel, but you can look it up in the code.

JSCanvas

The other class I would not want to mess around with much, because I am happy that it kind of works, is the JSCanvas class. It allows us to draw graphics inside a UI application. For this it uses the canvas tag from p5.js:

```
class JSCanvas extends JSObject {
    constructor() {
        super();

        // get the p5js canvas:
        this.element = document.getElementById("defaultCanvas0");
        this.element.style.padding = DEFAULT_PADDING;
        this.element.style.margin = DEFAULT_MARGIN;
        this.element.style = 'border: 1px solid green;';

        this.gobjects = [];
```

```
    }
    ...
}
```

As you see it gets a handle on p5.js's canvas, which happens to be called "defaultCanvas0". This is not really documented, and as soon as p5.js would change the name, this wouldn't work anymore. You also see the array of *gobjects*, and most of the methods are one-to-one equivalents to the graphics program functions. One things that is different, however, is the *updateJSCanvas()* function, take a look if you are interested.

.

Util Library

What is the point of the Util library? Well, as I stated before, I wanted to transfer my existing Java code to JavaScript with as little changes as possible. For the standard Java classes, I was first considering to use one of a transpilers like J2CL [1], JSweet [2], or the old GWT [3]. But the JavaScript code these transpilers generate is ugly as hell, and humongous in size. Since I needed just a handful of classes, I decided to write them myself, as there are:

- RandomGenerator
- StringTokenizer
- ArrayList
- HashMap
- HashSet
- FileReader
- FileWriter
- StorageReader
- StorageWriter
- URLReader

The last three do not exist in Java, but lend themselves to the use in a browser. Again, here ES6 came in real handy, and what other people might call syntactic sugar, I call a life safer.

The classes are really trivial. The RandomGenerator is a wrapper around Math.random(), the StringTokenizer a wrapper around JavaScripts string functions, and ArrayList, HashMap, and HashSet wrap their corresponding JavaScripts classes Array, Map and Set.

Similarily, FileReader is a wrapper around an XMLHttpRequest, and FileWriter is an <a> tag, stupid as it may sound, but works. One word on the FileReader: in ES6, there exists a class with the name FileReader in the new file API, hence we had to call ours Utils.FileReader, which on the other hand allowed us to introduce the concept of what we call *namespace*. Actually, all the different libraries should be made namespaces, but that would reduce the simplicity of use of our libraries by quite a bit.

include()

I wanted to call it *import*, but that name was taken. The idea was to import JavaScript classes as you can in Java using the import statements at the top of a Java file. Usually to import another JavaScript file you have to do this in the HTML header:

```
<html>
<head>
  <script src="./libraries/utils.js"></script>
  <script src="./ticTacToeLogic.js"></script>
</head>
...
```

But I did not want to have to modify the HTML file everytime I wanted to just import a little class. As usual the internet has the answers to all your questions:

```
/* https://www.educative.io/answers/how-to-dynamically-load-a-js-file-in-javascript */
function include(file, async = false) {
    let script = document.createElement("script");
```

```
    script.setAttribute("src", file);
    script.setAttribute("type", "text/javascript");
    script.setAttribute("async", async);

    document.body.appendChild(script);

    // success event
    script.addEventListener("load", () => {
        //console.log("File loaded")
    });
    // error event
    script.addEventListener("error", (ev) => {
        console.log("Error loading file", ev);
    });
}
```

Now all you have to do, add this line at the top of you code,

```
include("Pr4_Agrar/ticTacToeLogic.js");
```

and you can use your own handwritten classes.

.

[1] J2CL, https://github.com/google/j2cl
[2] JSweet: http://www.jsweet.org/
[3] GWT: https://padlet.com/lofidewanto/gwtintro
[4] FileAPI, https://w3c.github.io/FileAPI/

.

Karel Programs

Last, but not least we come to Karel. It is how we start to learn programming, and hence you would think it is the easiest to implement, but quite the opposite.

The core problem is that you do not want the code to run through at once, but instead, you want one line to be executed, then wait a second, then go to the next line, wait a second, and so on. JavaScript does not have a pause() method. And while doing so, you want to see the actual state and position Karel is in. JavaScript is not multithreaded. And in addition you have if-conditions and for- and while-loops in the Karel code, which makes simulating it a difficult task. Here JavaScript does have a very attractive feature: the eval() function. So, how do you circumvent the problems, and use the features?

From our point of view Karel programs are simply graphics programs. Karel is represented by four GImages, one for each direction he can move to. The p5.js draw() function looks like this:

```
function draw() {
    eval(codeLines[codePointer]);
    codePointer++;
    if (codePointer >= codeLines.length) {
        noLoop();
    }
    update();
}
```

We basically set the framerate to one frame per second, and every second we execute one line of Karel code, which has been parsed into an array. This actually solves our first and second problem.

For each of the Karel commands, like *move()* or *putBeeper()*, we have a function. For instance, for move it looks like this:

```
function move() {
    if (karelIsAlive) {
        if (frontIsClear()) {
            switch (karelDir) {
                case 0:
                    // this.karel.x += SIZE;
                    karel.move(SIZE, 0);
                    break;
...
}
```

That is what allows us to simply use JavaScript's eval() function.

The one problem that remains, are infinite loops. Hypothetically, one could just ignore the problem: it does not happen often, and when it does, you just restart the browser. But from a didactic point of view, and that is what Karel is all about, this is not very helpful.

However, one can tackle the problem. If we realize, that it is basically only the while-loops that can be problematic, we only need to look at those. One could try a static code analysis of while loops. To me, however, this looks quite a bit like the famous halting problem (I said Karel is tricky). Instead, we insert a little piece of code after every while we encounter. The following two lines of regular expression magic do that:

```
const regex = /(while\s*)\(([ a-zA-Z\(\)]+)\)\s*\{/g;
code = code.replaceAll(regex, " $1 ( $2 ) { catchInfiniteLoops();");
```

The *catchInfiniteLoops()* function is a simple counter:

```
let infiniteLoopCounter = 0;
function catchInfiniteLoops() {
    infiniteLoopCounter++;
    if (infiniteLoopCounter > 200) {
        throw new InfiniteLoopError("Most likely infinite loop!");
    }
}
```

We simplify our halting problem: if this method has been called more than 200 times, we say we have an infinite loop and throw an error. Thus executing the code in a simulation run with eval(code), will always stop, even if there is an infinite loop. And we can tell our users that their code most likely has an infinite loop.

There are a few more code gems in Karel (or hacks as some might call them), but you have to discover them for yourself!

.

.

Copyright © 2016-2023 Ralph P. Lano. All rights reserved.