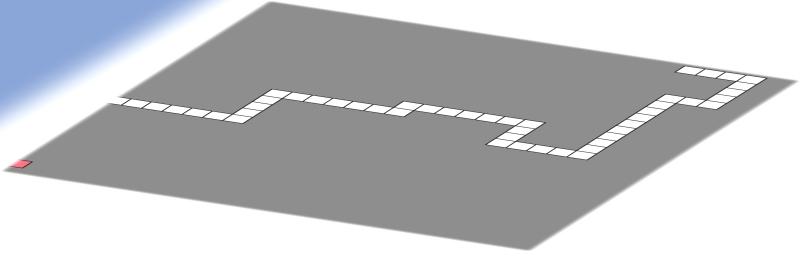


```

    // berechnet die neue Position aller Segmente
    move() {
        // neue Position für alle Schwanz Segmente (außer das vorderste Segment)
        forVar i=tail.length-1; i>0; i-{
            // aktuelles Segment bekommt Position vom Segment davor
            this.tail[i].col = this.tail[i-1].col;
            this.tail[i].row = this.tail[i-1].row;
        }
        // neue Position für das vorderste Segment - bekommt die Position vom Kopf
        this.tail[0].col = this.head.col;
        this.tail[0].row = this.head.row;
        if (this.direction === "RIGHT") {
            this.head.col++;
        }
        if (this.direction === "LEFT") {
            this.head.col--;
        }
        if (this.direction === "DOWN") {
            this.head.row++;
        }
        if (this.direction === "UP") {
            this.head.row--;
        }
    }
}

```



SIZZLE

V 1.0 DE

Dies ist ein Workshop in welchem wir das altbekannte Spiel „Snake“ programmieren werden. Bei diesem Klassiker aus den 80er Jahren muss man eine Schlange mit den Pfeiltasten durch ein Spielfeld steuern und dabei Futter aufsammeln. Dabei wächst die Schlange mit jedem Stück Futter. Mit der Zeit wird es schwierig die Schlange so zu steuern, dass sie sich nicht in den eigenen Schwanz beißt, denn dann wäre das Spiel zu ende.

Wie werden dieses Spiel als Web Browser Applikation entwickeln und dabei lernen in der Programmiersprache JavaScript zu programmieren. Wir benutzen dabei die JavaScript Bibliothek p5.js welche uns das Programmieren des Spielfeldes und der Steuerung der Schlange deutlich vereinfacht.

Daniel Wahl
daniel.wahl@bluewin.ch

EINFÜHRUNG	- 3 -
WERKZEUGE	- 3 -
EDITOR	- 3 -
INTERNET BROWSER	- 3 -
JAVASCRIPT	- 3 -
D5.JS	- 3 -
SIZZLE – LOS GEHT'S!	- 4 -
INDEX.HTML	- 4 -
SNAKE.JS	- 4 -
CANVAS	- 5 -
RECT()	- 6 -
SPIELE KONSTANTEN & VARIABLEN	- 7 -
DIE SNAKE KLASSE	- 8 -
AUFBAU & VERHALTEN	- 9 -
KLASSE SNAKE	- 9 -
DER KONSTRUKTOR	- 9 -
DIE KLASSE SEGMENT	- 10 -
MACH EIN NEUES SEGMENT	- 11 -
ERSTELLE EINE NEUE SNAKE	- 12 -
LERNE ZU ZEICHNEN	- 13 -
LERNE ZU KRIECHEN	- 16 -
ÄNDERE DEINE RICHTUNG ...	- 19 -
EVENT HANDLER	- 19 -
FUTTER	- 22 -
ESSEN	- 24 -
WACHSEN	- 26 -
SPIEL BEENDEN	- 27 -
SCHLUSSWORT	- 28 -
PROGRAMM CODE	- 29 -
INDEX.HTML	- 29 -
SNAKE.JS	- 30 -

EINFÜHRUNG

Das ist ein Programmierworkshop in welchem Du ein kleines Spiel auf dem Computer programmieren wirst. Der Workshop ist für Programmieranfänger konzipiert. Du solltest jedoch trotzdem gewisse Grundkenntnisse besitzen. Falls Dir Begriffe wie Variablen, Schleifen und Funktionen bereits etwas sagen, kannst Du ohne Probleme weiterlesen. Falls nicht, kannst Du erst den Artikel „Programmiergrundlagen“ lesen.

WERKZEUGE

EDITOR

Als Werkzeug dazu brauchst Du ein Computer und einen Text Editor. Ein ganz einfacher Editor wie Notepad oderTextEdit würde genügen. Ein Programm Code Editor macht das Programmieren allerdings einiges komfortabler. Einige sind gratis und Du kannst sie vom Internet runterladen. Einige gute Beispiele sind:

- [Atom](#)
- [Notepad++](#)
- [Microsoft Visual Code](#)
- [Komodo Edit](#)

INTERNET BROWSER

Wir werden das Spiel als Browergame entwickeln. Um das Programm zu starten brauchst Du deshalb einen Browser. Welcher ist unwichtig. Ich benutze gerne Google Chrome und werden einige Male einige Google Chrome spezifische Menus erläutern. Aber jeder andere Browser funktioniert genauso gut.

JAVASCRIPT

Wir werden das Spiel in der Programmiersprache JavaScript schreiben. Glücklicherweise versteht heutzutage jeder Browser JavaScript. Das heisst wir müssen nichts weiter dafür installieren.

D5.JS

Wir werden als Hilfe eine JavaScript Bibliothek mit dem Namen D5.js benutzen. D5.js wird uns einiges sehr vereinfachen mit der Programmierung von unserem Spiel. Wenn wir Funktionen aus der D5 Bibliothek benutzen werde ich dies jeweils erwähnen. Es lohnt sich aber auf jeden Fall mal auf die Webseite von [D5.js](#) zu schauen und nachzulesen was D5 alles kann und bietet.

Tatsächlich müssen wir um D5 zu in unserem Programm zu nutzen eine Datei herunterladen. Das kann aber unser Webbroweser erledigen und wir werden das entsprechend programmieren.

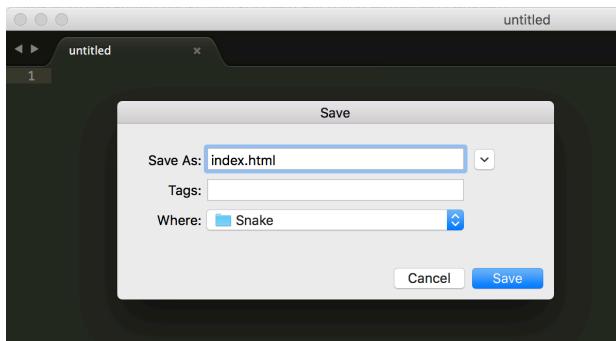
SIZZLE – LOS GEHT'S!

INDEX.HTML

So, wenn wir unseren Editor installiert haben können wir auch schon starten.

Lass uns erst mal ein Verzeichnis erstellen, wo wir unser Game Projekt speichern. In diesem Verzeichnis werden wir alle Dateien welche zu diesem Spiel gehören sammeln. Du kannst das Verzeichnis benennen wie du willst, zum Beispiel „snake“.

Im neuen Verzeichnis erstellen wir eine Datei. Das machst Du am besten mit Deinem Editor Editor. Der Name der Datei sollte **index.html** lauten.



In unsere index.html Datei kommt nur wenig Code. Eigentlich sind es nur zwei Instruktionen, dass der Browser zwei zusätzliche Dateien laden soll.

```
<html>
<head>
  <script language="javascript" type="text/javascript"
    src="https://p5js.org/assets/js/p5.min.js"></script>
  <script language="javascript" type="text/javascript" src="snake.js"></script>
</head>
</html>
```

Die erste Datei (<https://p5js.org/assets/js/p5.min.js>) ist die p5.js JavaScript Bibliothek welche uns die Programmierung erleichtern wird. Am src Attribut kann man erkennen, dass die Datei aus dem Internet geladen wird. Wir müssen sie also nicht von Hand herunterladen – das erledigt der Internetbrowser für uns.

Die zweite Datei (snake.js) enthält unser Programm und existiert momentan noch gar nicht. Falls Du nach dem Speichern der index.html Datei, diese in Deinem Browser öffnest, wirst Du nur eine weisse Seite sehen.

Als nächstes sollten wir die Datei **snake.js** erstellen und unser Spiel darin programmieren.

SNAKE.JS

Erstelle mit Deinem Editor eine neue Datei im selben Verzeichnis wie die index.html Datei und benenne sie **snake.js**.

Die D5 Bibliothek erwartet von unserem Programm zwei wichtige Funktionen:

setup() Wird von D5 einmal automatisch ausgeführt sobald das Programm im Browser geladen ist.

draw() Wird von D5 nach setup() immer wieder ausgeführt bis das Programm beendet ist.

Als erstes müssen wir in der snake.js Datei die Funktion **setup()** erstellen. Die **setup()** Funktion wird von D5 automatisch erkannt und als erstes ausgeführt.

```
function setup() {  
}
```

Eine JavaScript Funktion wird immer als solche mit dem Wort **function** eingeleitet, gefolgt von ihrem Namen. Man kann einer Funktion Werte (Zahlen oder Text) in Variablen übergeben. Diese würden dann innerhalb der runden Klammern () aufgelistet.

In den geschwungenen Klammern {} wird der Programm Code dieser Funktion geschrieben – und genau das werden wir jetzt tun.

PS: die geschwungenen Klammern sind wahrscheinlich auf Deiner Tastatur nicht aufgedruckt, aber Du kannst sie mit der Tastenkombination **ALT-8** und **ALT-9** schreiben.

CANVAS

In unserer setup() Funktion erstellen wir eine sogenannte Canvas (engl. Leinwand). In der Canvas wird sich unser Snake Spiel abspielen. Dazu benutzen wir die Funktion **createCanvas()**. **createCanvas()** ist eine Funktion der p5.js Bibliothek. Wir müssen allerdings noch zwei Werte übergeben, welche die Grösse unserer Canvas definieren. Unsere Cavas soll 600 auf 600 Pixel gross sein, also geben wir diese Werte als Zahlen mit Komma getrennt in den runden Klammern mit.

```
function setup() {  
    createCanvas(600, 600);  
    background(50);  
}
```

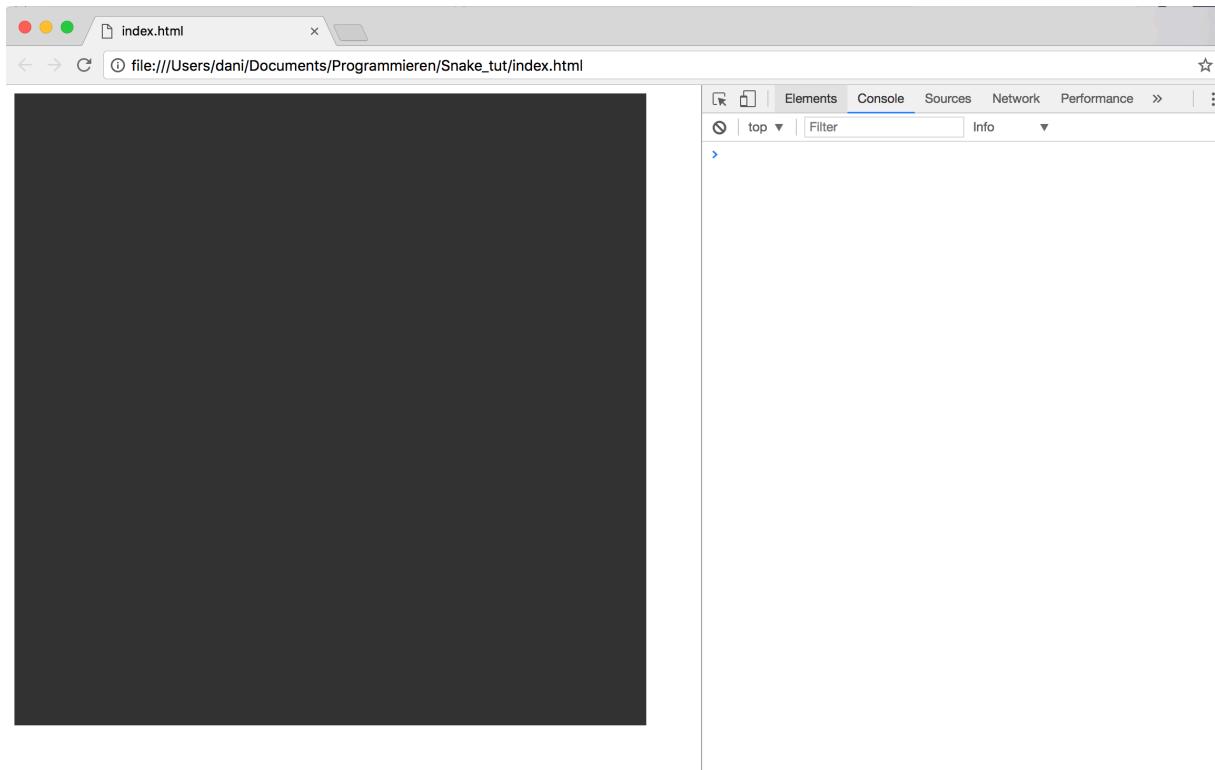
Wir setzen mit dem Befehl **background(50)** auch gleich die Hintergrundfarbe auf einen dunkelgrauen Wert.

Bitte achte darauf, dass alle Befehle mit einem Semikolon (;) abgeschlossen werden.

Wenn wir jetzt zum Browser wechseln und die Seite index.html neu laden, sollten wir ein dunkelgraues Quadrat – unsere Canvas – sehen. Gratulation! Wenn das funktioniert, hast Du bereits unser Spielfeld programmiert.

Falls das nicht funktioniert, solltest du den Programmcode nochmals genau überprüfen. Man vergisst leicht ein ; am Ende eines Befehls oder vertut sich mit den geschwungenen Klammern.

Du kannst als Hilfe im Browser die Entwickertools einschalten. (Im Google Chrome Browser im Menu Anzeigen – Entwickler – JavaScript-Konsole). Hier werden Fehler im Programm angezeigt. Es ist ein sehr gutes Hilfsmittel um zu erkennen wo man einen Fehler gemacht hat.



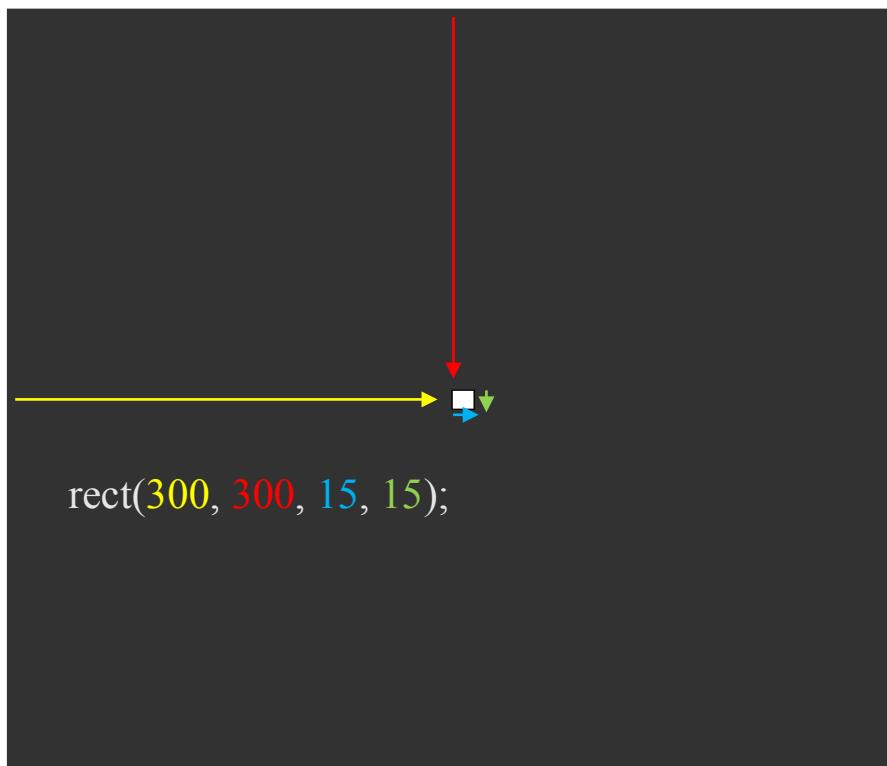
RECT()

Als nächstes zeichnen wie ein kleines, weisses Quadrat in die Mitte unserer Canvas. So soll nämlich später unsere Schlange gezeichnet werden.

Wir können den Befehl für das Quadrat für jetzt mal direkt in die setup() Funktion schreiben.

```
function setup() {  
    createCanvas(600, 600);  
    background(50);  
    rect(300, 300, 15, 15);  
}
```

Die Funktion um ein Quadrat (oder Rechteck) zu zeichnen lautet rect(), und erwartet einige Werte innerhalb der runden Klammern. Die ersten zwei Werte beschreiben die Position innerhalb der Canvas und zwar die Anzahl Pixel von links und von oben. Die dritte und vierte Zahl beschreibt die Grösse des Rechteckes, wieder in Anzahl Pixel nach rechts und nach unten.

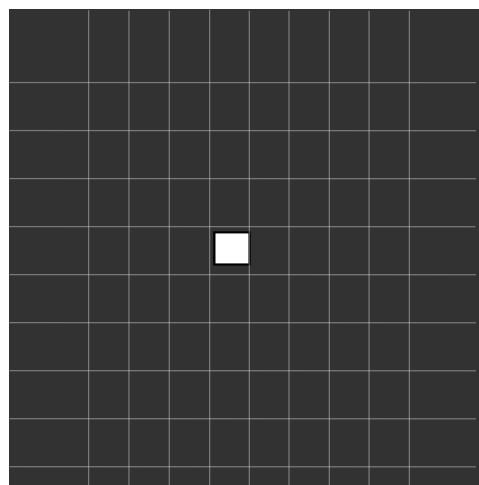


Wie Du siehst kann man mit der Funktion rect() sowohl Rechtecke als auch Quadrate zeichnen. Du kannst die Zahlen im Programmcode variieren und im Browser anschauen wie sich die Änderungen auswirken. Dazu musst Du die Datei snake.js speichern und den Browser die Seite neu laden.

SPIELE KONSTANTEN & VARIABLEN

Als erstes sollten wir uns überlegen wie wir unsere Schlange durch die Canvas steuern. Unser Programm soll die Schlange mit Hilfe der ersten beiden Werten von rect() steuern, wie wir gelernt haben können wir ja so die Position eines Quadrates auf der Canvas bestimmen. Wir wollen aber die Schlange nicht Pixel um Pixel steuern. Die Schlange und ihr Futter sollen sich in einem fixen Gitter von Zeilen und Kolonnen über die Canvas bewegen.

Die Anzahl dieser Kolonnen und Zeilen in der Canvas, sowie die Grösse des Gitters wird sich während des Spieles nicht ändern, bleiben also konstant. Wenn sich ein Wert während der „Lebensdauer“ eines Programmes nicht ändert sollte man diese in einer Konstante speichern. Das werden wir als nächstes tun.



Wir schreiben also unsere Konstanten wie Anzahl und Grösse des Gitters als Konstanten mittels des **const** Befehls zuoberst in unseren Programm Code in die Datei snake.js.

Bitte beachte: Die Definition der Konstanten sollte nicht innerhalb der Funktion `setup()` stehen, sondern oberhalb.

```
const COLS=40;
const ROWS=40;
const FIELD_SIZE=15;
```

Wir legen also fest, dass wir in unserer Canvas 40 auf 40 Felder haben, und ein Feld eine Seitenlänge von 15 Pixeln hat.

Variablen welche Werte enthalten die sich im Verlauf des Programms ändern werden mit dem Befehl `let` definiert. Dazu werden wir später noch kommen.

Nun sollten wir noch die Befehle in der `setup()` Funktion so anpassen, dass anstatt der Zahlen welche wir dort eingetragen haben unsere Konstanten verwendet werden:

```
function setup() {
    createCanvas(COLS*FIELD_SIZE, ROWS*FIELD_SIZE);
    background(50);
    rect(width/2, height/2, FIELD_SIZE, FIELD_SIZE);
}
```

Schauen wir uns nochmals an was wir hier genau gemacht haben.

Wir haben entschieden, dass unsere Canvas in 40 Kolonnen und 40 Zeilen unterteilt werden soll, und dass diese Felder je eine Grösse von 15 Pixeln haben soll. Da diese Größen sich während dem Spiel nicht ändern haben wir diese in Konstanten geschrieben.

Diese Konstanten haben wir nun in der `setup()` Funktion benutzt um die Canvas zu erstellen. Dazu haben wir Feldgröße mit der Anzahl Kolonnen oder Zeilen multipliziert (mit dem *). Wenn wir 40 x 15 Pixel rechnen kommen wir nämlich wieder auf unsere 600 Pixel welche die Canvas ursprünglich hatte.

Wenn wir uns jetzt irgendwann entscheiden, dass unsere Canvas doch grösser oder kleiner sein soll, können wir jetzt einfach die Werte in diesen Konstanten ändern.

Aber den `rect()` Befehl hat sich auch geändert. Schauen wir uns das auch nochmals an.

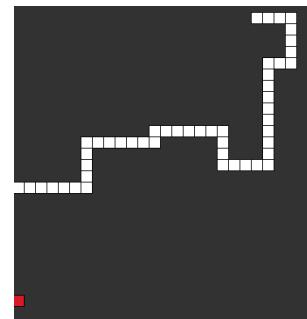
Als erstes haben wir die Position des Quadrates von **300, 300** auf **width/2, height/2** geändert. Dazu muss man wissen, dass die Funktion `createCanvas()` welche wir ja bereits aufgerufen haben, uns zwei Variablen generiert. Diese heissen **width** und **height** und beinhalten die Anzahl Pixel der Breite und Höhe der Canvas. Wenn wir unsere Schlange in der Mitte der Canvas positionieren wollen können wir die **width** und **height** Variablen benutzen und diese durch 2 teilen. So bleibt sie Startposition in der Mitte, auch wenn wir in Zukunft die Grösse der Canvas mittels unserer Konstanten ändern.

DIE SNAKE KLASSE

So, nun sollten wir uns endlich einmal an die Arbeit machen unsere Schlange zu entwickeln. Bevor wir mit dem Programmieren loslegen können müssen wir uns aber überlegen wie unsere Schlange aufgebaut ist und was sie können soll.

AUFBAU & VERHALTEN

- Unsere Schlange besteht aus einem Kopf und einem Schwanz.
- Kopf sowie Schwanz bestehen aus quadratischen, jeweils 15 x 15 Pixel grossen Segmenten.
- Die Länge des Schwanzes soll wachsen können indem zusätzliche Segmente hinzugefügt werden.
- Der Kopf der Schlange lässt sich mit den Pfeiltasten nach oben, unten, links oder rechts steuern.
- Der Kopf der Schlange bewegt sich so lange in eine Richtung, bis die Richtung mit einer Pfeiltaste geändert wird.
- Jedes Segment des Schwanzes folgt der Bewegung des Vorhergehenden.
- Die Schlange kann Futter fressen, dabei wird der Schwanz um ein Segment länger.
- Beisst sich die Schlange in den eigenen Schwanz, ist das Spiel zu ende.
- Verlässt die Schlange das Spielfeld, ist das Spiel zu ende.



KLASSE SNAKE

Wir werden unsere Schlange Objekt Orientiert programmieren.

Eine Klasse ist wie ein Bauplan von einem Ding – oder einem Objekt. Mit dem Bauplan kann man ein Objekt erstellen. Ähnlich ist es beim Programmieren. Wir programmieren eine Klasse und beschreiben dem Computer wie er das Objekt erstellen soll und wie es funktioniert.

Also, legen wir los und erstellen erst mal eine leere Klasse mit dem Namen **Snake**.

```
class Snake {  
}
```

Innerhalb der Klasse Snake werden wir als nächstes Funktionen und Variablen welche zur Klasse Snake gehören hinzufügen.

DER KONSTRUKTOR

Die erste Funktion welche wir programmieren müssen ist die Konstruktor Funktion. Jede Klasse aus welcher ein Objekt gemacht werden soll braucht einen Konstruktor, diese wird ausgeführt, wenn wir von der Klasse ein Objekt erstellen. Darin steht welche Variablen das Objekt haben soll.

```
class Snake {  
  
    constructor() {  
        this.direction = "RIGHT";  
        this.head = null;  
        this.tail = [];  
    }  
}
```

So, unsere Snake Klasse hat nun einen Konstruktor in dem wir drei Variablen definiert haben. Wenn wir ein Objekt der Klasse Snake erstellen hat dieses Objekt die drei Variablen **direction**, **head** und **tail**.

Wie Du siehst haben wir den Variablen Namen das Wort **this**. vorangestellt. Das ist wichtig damit der Computer versteht, dass die Variabel zu diesem Objekt gehört. Variablen welche zu einem Objekt

gehören haben auch einen speziellen Namen, man spricht dann von einem Attribut. Ein Attribut ist eine Eigenschaft von einem Objekt.

Ein Beispiel:

Ich heisse Dani und bin ein Objekt der Klasse Mensch. Ich habe wie alle Menschen verschiedene Attribute wie Grösse, Haarfarbe, Augenfarbe usw. Von der Klasse Mensch gibt es aber viele Objekte und diese haben dieselben Attribute wie ich (jeder hat eine Grösse, eine Haarfarbe und eine Augenfarbe). Nur die Werte dieser Attribute unterscheiden sich.

Sehen wir uns doch die Attribute unserer Snake Klasse an:

```
this.direction = "RIGHT";
```

Da wir unsere Schlange in eine Richtung steuern wollen muss die Schlange wissen wo sie sich hinbewegen soll. Das wird im Attribut **direction** gespeichert. Beim Erstellen eines Objektes vom Typ Snake, soll die Richtung auf Rechts (right) gesetzt werden.

```
this.head = null;  
this.tail = [];
```

Da unsere Schlange einen Kopf und einen Schwanz braucht, werden wir diese in den Attributen **head** und **tail** speichern. Kopf sowie Schwanz bestehen aus quadratischen, jeweils 15 x 15 Pixel grossen Segmenten. Kopf brauchen wir nur einen. Für den Schwanz brauchen wir viele Segmente, deshalb haben wir das Attribut **tail** als **Array** definiert.

Ein Array ist eine Liste von Dingen. In ein Array kann man eine Liste von Nummern, Text oder eben Schlangensegmente versorgen.

Unser Array tail ist noch leer, deshalb steht in den eckigen Klammern auch noch nichts. Auch der Kopf der Schlange besteht noch aus nichts – deshalb hat das Attribut den Wert null.

Wir müssen wir nun unsere Segmente programmieren. Diese sollen auch als Klasse entwickelt werden.

DIE KLASSE SEGMENT

Wir fügen unterhalb unserer Snake Klasse eine neue Klasse Segment ein und versehen diese gleich mit einem Konstruktor.

```
class Segment {  
  
    constructor (start_col, start_row) {  
        this.col = start_col;  
        this.row = start_row;  
        this.size = FIELD_SIZE;  
        this.color = color(255);  
    }  
}
```

Der Konstruktor hat in seinen runden Klammern die zwei Variablen **start_col** und **start_row**. Diese beschreiben das Spielfeld an welchem das Segment Objekt sich befinden soll, und werden im Konstruktor auf die Attribute **this.col** und **this.row** übergeben. So können wir die Kolonne und Reihe für ein Objekt festlegen gleich, wenn es erstellt ist.

Das Attribut **size** (die Grösse des Segments) wollen wir nicht jedes Mal wenn wir ein neues Segment erstellen erwähnen müssen, da wir ja dafür eine Konstante definiert haben.

Wir wollen dem Segment noch eine Farbe mitgeben. Diese soll vorerst mal weiss sein, darum bekommt das Attribut **color** den Wert **color(255)**. **color()** ist wieder eine Funktion aus der p5.js Bibliothek welche uns das Programmiererleben vereinfacht.

MACH EIN NEUES SEGMENT

So, mit unser neuen Klasse Segment können wir nun endlich den Konstruktor unserer Klasse **Snake** vervollständigen und ihr einen Kopf geben und Schwanz.

```
class Snake {  
  
    constructor() {  
        this.direction = "RIGHT";  
        this.head = new Segment(COLS/2, ROWS/2);  
        this.tail = [];  
  
        let ts1 = new Segment(this.head.col-1, this.head.row);  
        this.tail.push(ts1);  
    }  
}
```

Das Attribut **this.head** bekommt nun im Konstruktor ein neues Segment. Das **new** ist nämlich die Syntax um von einer Klasse ein Objekt zu erstellen. Wie im Konstruktor der Segment Klasse beschrieben können wir gleich beim Erstellen eines Segment Objekts angeben wo sich das neue Objekt befinden soll. Da wir unsere Schlange in der Mitte des Spielfeldes starten lassen wollen benutzen wir unsere Konstanten für die Anzahl Kolonnen und Zeilen und berechnen mit **COLS/2** und **ROWS/2** das mittlere Feld.

Für das Attribut **this.tail** erstellen wir erst wieder ein neues Segment Objekt und speichern dies in einer Variabel mit dem Namen **ts1**. Wir wollen das neue Schwanzsegment links vom Kopf platzieren denn unsere Schlange soll ja irgendwann nach rechts loskriechen. Deshalb benutzen wir die **col** und **row** Attribute vom Schlangenkopf und verschieben das Schwanz Segment eine Position nach links (**col-1**).

Das erscheint vielleicht ein wenig kompliziert. Lass uns deshalb nochmals überlegen was **this.head.col-1** im Constructor von Snake tut:

Wir wollen ein neues Segment – das erste Segment vom Schwanz der Schlange erstellen. Dieses Segment soll im Feld links vom Kopf der Schlange gezeichnet werden.

Wir haben in der zweiten Zeile des Snake Konstruktors ein Objekt vom Type Segment erstellt und es im Attribut **head** gespeichert:

this.head = new Segment(COLS/2, ROWS/2);

Das erstellte Segment Objekt enthält wiederum das Attribut **col** in welchem eine Zahl gespeichert ist – nämlich die Nummer der Kolonne in welcher der Schlangenkopf gezeichnet werden soll.

Da wir den Schlangenschwanz links vom Kopf zeichnen wollen, können wir die selben Koordinaten vom Kopf nehmen, müssen aber die Kolonne um 1 subtrahieren. Genau das passiert in **this.head.col-1**, wir nehmen die Zahl von der Kolonne **col** im Objekt **head** und subtrahieren 1.

Den errechneten Wert benutzen übergeben wir in den Konstruktor des neuen Segmentes.

Sobald das neue Segment erstellt ist, fügen wir es mit der Array Funktion **push()** dem tail Array hinzu.

ERSTELLE EINE NEUE SNAKE

Als nächstes gehen wir zurück in die **setup()** Funktion und machen 2 Änderungen:

1. Wir entfernen den **rect()** Befehl. Wir wollen, dass unsere Schlange sich selber zeichnen kann.
2. Wir erstellen endlich eine Schlange – ein Snake Objekt.

```
const COLS=40;
const ROWS=40;
const FIELD_SIZE=15;
let sizzle;

function setup() {
  createCanvas(COLS*FIELD_SIZE, ROWS*FIELD_SIZE);
  background(50);
  sizzle = new Snake();
}
```

Wenn wir ein Snake Objekt erstellen sollten wir dieses in einer Variabel speichern. Diese Variabel sollte im gesamten Programm verfügbar sein. Deshalb müssen wir die Variabel für unsere Schlange auch ausserhalb der **setup()** Funktion anlegen. Das machen wir direkt unterhalb der **const** Deklarationen bei welchen wir unsere Konstanten definiert haben. Unsere Schlange sollte einen Namen haben, nennen wir unsere Schlange **sizzle**.

In der **setup()** Funktion wird dann das Schlangen Objekt endlich erstellt und in der Variabel **sizzle** gespeichert.

Juhuu! Unsere **sizzle** lebt. Aber, ... wenn wir die Seite im Browser neu laden haben wir wieder nur eine graue Fläche. Das kleine Quadrat ist verschwunden.

Wir habe ja in der **setup()** Funktion den **rect()** Befehl entfernt. Deshalb ist das Quadrat verschwunden. Ausserdem haben wir ja erst ein Snake Objekt erstellt, haben ihm aber noch nicht beigebracht wie es sich auf unserer Canvas zeichnen soll.

Bevor wir das tun wollen wir aber doch noch überprüfen ob unsere Schlange wirklich existiert. Dazu öffnen wir im Browser die Konsole wenn wir sie nicht schon offen haben (Menu Anzeigen – Entwickler - JavaScript-Konsole). In der Konsole tippen wir den Namen unserer Schlange ein. Da **sizzle** eine globale Variabel ist (wir haben sie ausserhalb aller Funktionen mit **let** definiert), ist sie auch für die Browser Konsole sichtbar. Wenn wir **sizzle** tippen und mit Return bestätigen gibt uns die Konsole das **sizzle** Objekt aus:

The screenshot shows the Chrome DevTools Console tab. It displays the following object structure:

```
> sizzle
< ▾ Snake {direction: "RIGHT", head: Segment, tail: Array(1)} ⓘ
  direction: "RIGHT"
  ▾ head: Segment
    col: 20
    color: d.Color
    row: 20
    size: 15
    ▶ __proto__: Object
  ▾ tail: Array(1)
    ▾ 0: Segment
      col: 19
      color: d.Color
      row: 20
      size: 15
      ▶ __proto__: Object
      length: 1
      ▶ __proto__: Array(0)
      ▶ __proto__: Object
```

Wie wir in der Konsole sehen können wurde unser Objekt vom Typ **Snake** angelegt.

Das Attribut **head** ist vom Typ **Segment** und hat seine eigenen Attribute **col**, **row**, **size** und **color**.

Das Attribut **tail** ist ein **Array** mit einem Element vom Typ **Segment**, welches sich an der Position **col: 19** und **row: 20** befindet.

LERNE ZU ZEICHNEN

Wir könnten nun eine Funktion schreiben welche alle Segmente von sizzle anschaut und an den entsprechenden Positionen ein Quadrat zeichnet. Das wäre zwar möglich aber entspräche nicht der Objekt Orientierten Programmierung.

Wir wollen die Logik für das Zeichnen der Segmente in der entsprechenden Klasse selber unterbringen - wir wollen der Schlange selber beibringen sich zu zeichnen.

Wir fügen der **Klasse Segment** also eine Funktion hinzu, welche wir aufrufen können damit sich das Segment zeichnet. Übrigens, Funktionen einer Klasse haben wie Attribute einen speziellen Namen – man nennt sie **Methoden**.

Wir schreiben also (unterhalb des Konstruktors) eine neue Methode mit dem Namen **draw()**. Denn innerhalb jedes Objektes vom Typ Segment haben wir Zugriff auf die Attribute **col**, **row**, **size** und **color** – alles was wir für das Zeichnen eines Quadrates brauchen

```
draw() {
  fill(this.color);
  rect(this.col*this.size, this.row*this.size, this.size, this.size);
}
```

Mit dem Befehl **fill()** setzen wir die Farbe mit welcher wir das Quadrat ausfüllen wollen. Wir übergeben **fill()** noch einen Farbwert (255=weiss den wir im Konstruktor von Segment definiert haben).

Den Befehl **rect()** kennen wir ja schon. Wie müssen aber daran denken, dass **rect()** als erste zwei Werte die Pixel Koordinaten in der Canvas will. Wir haben aber in **col** und **row** von Segment nicht Pixel sondern Koordinaten unserer Spielfelder gespeichert. Wir können aber die Pixel Koordinaten berechnen indem wir **col** und **row** jeweils mit der **FIELD_SIZE** multiplizieren. Da wir das Segment mit der **FIELD_SIZE** erstellt haben ist es im Attribut **size** gespeichert: **this.col * this.size** ergibt also die korrekte Anzahl Pixel von der linken Seite aus, wo sich unser Segment zeichnen soll.

Nun haben wir zwar eine Methode um Segmente zu zeichnen, wir müssen diese aber noch aufrufen. Dazu schreiben wir eine neue Methode **drawSegments()** in der Klasse **Snake** welche über alle ihre Segmente Bescheid weiss und diese zeichnen soll.

```
drawSegments() {  
    this.head.draw();  
}
```

Innerhalb von **Snake** rufen wir das Attribut **this.head** und befehlen ihm die Methode **draw()** auszuführen. Da **this.head** ja vom Typ **Segment** ist und wir der Klasse **Segment** gerade beigebracht haben wie man ein Quadrat zeichnet sollte das funktionieren.

Doch leider sehen wir immer noch kein Quadrat wenn wir die Seite neu laden. Das liegt daran, dass die Methode **sizzel.drawSegments()** noch gar nirgends aufgerufen wird, und diese ruft ja wiederum **draw()** vom Segment.

Wie auf Seite - 4 - erwähnt erwartet D5 die zwei Funktionen **setup()** und **draw()** und, dass **draw()** während das Programm läuft immer wieder aufgerufen wird. Unser Programm soll die Position von Sizzle immer wieder neu berechnen und danach neu zeichnen. Darum ist **draw()** der geeignete Platz um **drawSegments()** aufzurufen. Lass uns **draw()** direkt unterhalb von **setup()** schreiben.

```
function draw() {  
    sizzel.drawSegments();  
}
```

Wenn wir jetzt den Browser neu laden lassen sehen wir endlich unser gezeichnetes **sizzel.head** Segment.

Nun sollten wir aber auch noch dafür sorgen, dass der Schlangenschwanz auch gezeichnet wird. Momentan hat das **sizzel.tail** Array ja erst ein Segment, aber im Verlauf des Spiels werden ja mehr und mehr Segmente hinzugefügt. Deshalb müssen wir die Segmente des Schwanzes in einer Programm Schleife abarbeiten.

Wir ändern also die **drawSegments()** Methode folgendermassen ab:

```
drawSegments() {  
    this.head.draw();  
  
    for (let i=0; i < this.tail.length; i++)  
        this.tail[i].draw();  
}
```

OK, damit das Sinn macht müssen wir erst mal etwas über Arrays lernen.

Wie bereits besprochen, ist ein Array quasi eine Liste in welcher man viele Variablen oder Objekte versorgen kann. Im Snake Konstruktor haben wir das **tail** Attribut als leeres Array angelegt, und haben dann ein Objekt vom Typ **Segment** darin abgelegt. Im Verlauf vom Spiel werden viele zusätzliche Segment Objekte in diesem Array hinzugefügt.

Jedes einzelne Element in einem Array erhält eine Nummer – die sogenannte Index Nummer.

Wenn wir in der Browser Konsole das **sizzel** Objekt anschauen, sehen wir, dass das **tail** Array die Länge (**length**) 1 hat und, dass das 0-te Element unser erstes Segment ist. 0 ist die **Index Nummer** des ersten Elementes. Wenn wir Elemente aus einem Array ansprechen wollen ist es wichtig zu verstehen, dass die Nummerierung bei **0** und nicht bei **1** beginnt.

```
▼ tail: Array(1)  
  ► 0: Segment  
    length: 1
```

Wenn wir also das erste (und momentan einzige) Segment aus sizzle.tail ansprechen wollen um zB seine draw() Methode aufzurufen müssten wir **sizzle.tail[0].draw()**; schreiben. Die Index Nummer kommt dabei in eckige Klammern. (Die eckigen Klammern findest Du auf einer Schweizer Tastatur mit der Tastenkombination **ALT-5** und **ALT-6**).

Da wir ja nicht alle Elemente im tail Array einzeln anschreiben wollen programmieren wir eine for – Schleife welche sich für alle Elemente im Array wiederholt. Schauen wir uns die Details der for – Schleife genauer an. Es ist wichtig, dass wir verstehen was hier passiert, da wir for – Schleifen immer wieder benutzen werden.

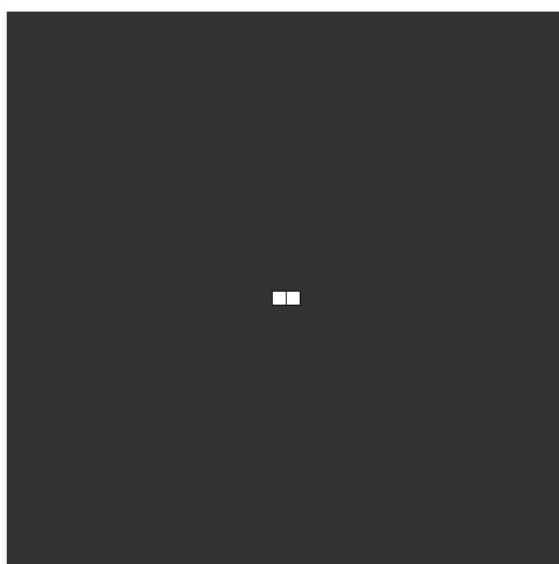
```
for (let i=0; i < this.tail.length; i++) {  
    this.tail[i].draw();  
}
```

Das Schlüsselwort **for** leitet die Schleife ein. Innerhalb der runden Klammern gibt es drei Teile:

let i=0;	Wir definieren eine Variabel mit dem Namen i und geben ihr den Wert 0
i < this.tail.length;	Wir vergleichen i mit der Länge vom Array this.tail . Die Schleife wird so lange wiederholt wie i kleiner ist als this.tail.length
i++	Erhöht den Wert von i um 1 nach jedem durchlaufen der Schleife.

Der Programmcode der innerhalb der geschwungenen Klammern **{ }** steht wird mit jedem durchlaufen der Schleife ausgeführt. Da wir in unserer Schleife das Array **this.tail[i]** mit der Variabel **i** abfragen, und diese Variabel von der Schleife ja jedes Mal um 1 erhöht wird können wie so die Methode **draw()** von jedem einzelnen Segment aufrufen.

Wenn wir die Seite im Browser neu laden sehen wir tatsächlich zwei Quadrate von sizzle, das Segment vom Kopf und links davon das momentan einzige vom Schwanz.



Im den nächsten Kapiteln werden wir lernen wie wir unserer sizzle beibringen zu kriechen.

LERNE ZU KRIECHEN

Es wird Zeit, dass sich sizzle bewegen kann. Um das zu ermöglichen müssen wir die Positionen aller Segmente verändern. Wir sollten mit dem Kopf beginnen, denn der weiss ja wo es langgeht.

Zuerst sollten wir der Klasse Snake eine neue Methode mit dem Namen **move()** schreiben. Darin soll die ganze Logik der Bewegung programmiert werden.

```
move() {  
}
```

So, die Schlange hat ja das Attribut **direction** welches wir beim Erstellen auf „RIGHT“ gesetzt haben. Wir werden **direction** benutzen um die **col** und **row** Attribute so zu ändern, dass der Kopf von sizzle in die richtige Richtung kriecht.

Dazu werden wir eine Reihe von Entscheidungen schreiben bei der wir den Wert von **sizzle.direction** prüfen. Das tun wir mit einem **if** – Block.

```
move() {  
    if (this.direction === 'RIGHT') {  
        this.head.col++;  
    }  
}
```

Der if – Block ist ähnlich aufgebaut wie die for – Schleife. In den runden Klammern prüfen wir, ob das Attribut **this.direction** den Wert **RIGHT** hat. Der Programm Code in den geschwungenen Klammern wird nur dann ausgeführt, wenn die Prüfung wahr ergibt, die **direction** also tatsächlich **RIGHT** ist. In unserem Fall wird dann das **col** Attribut vom Segment **this.head** um 1 erhöht. (Wir haben das **++** für eine numerische Variabel schon in der for – Schleife gesehen. Das funktioniert natürlich auch bei anderen Variablen, solange sie eine Zahl als Wert haben).

Jetzt müssen wir uns nur noch überlegen wann wir die Methode **move()** aufrufen wollen. Richtig, **move()** sollte in die **draw()** Funktion, nach dem Befehl **sizzle.drawSegments()**. Wir wollen ja, dass sich Sizzle auf unserer Canvas zeichnet und dann ihre neue Position berechnet – und das wollen wir immer wiederholen.

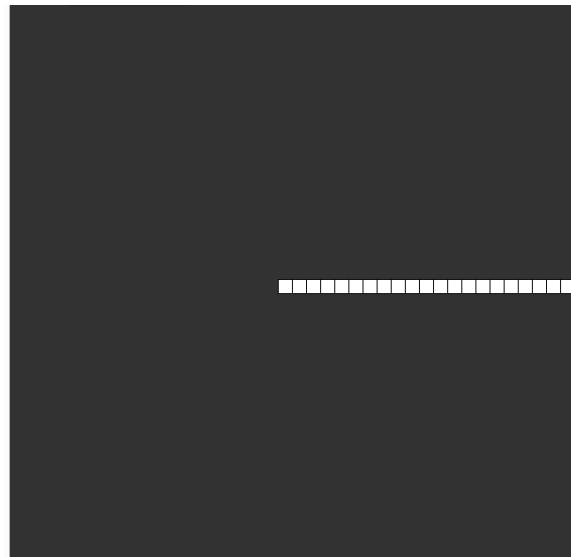
```
function draw() {  
    sizzle.drawSegments();  
    sizzle.move();  
}
```

Wenn wir das Programm im Browser neu laden, sollte unsere sizzle wieder gezeichnet werden, zudem düst sie gleich nach dem laden nach rechts aus dem Bild.

Es gibt noch einige Probleme um die wir uns kümmern müssen.

1. Sizzle ist ja nur 2 Segmente lang, hinterlässt aber eine lange Spur auf der Canvas. Der Grund dafür ist, dass wir ja immer neue Quadrate auf die Canvas zeichnen aber die alten nicht entfernen.
2. Das Spiel läuft viel zu schnell. Das ist weil die Funktion **draw()** viel zu schnell hintereinander ausgeführt wird.
3. Das dritte Problem ist zwar noch nicht sichtbar, aber momentan bewegt sich nur der Kopf, der Schwanz bleibt einfach stehen.

Wie sollten diese Probleme zuerst angehen bevor wir uns um die Steuerung kümmern.



Das **Problem Nr. 1** lässt sich einfach lösen. Wenn wir uns die **setup()** Funktion anschauen sehen wir, dass wir mit dem Befehl **background(50)** die Canvas dunkelgrau ausmalen. Wir können **background(50)** in die **draw()** Funktion verschieben. Dann wird die Canvas nicht nur am Anfang, sondern bei jedem Schritt der Schlange neu gezeichnet – und alle alten Quadrate dabei übermalt.

```
function setup() {
    createCanvas(COLS*FIELD_SIZE, ROWS*FIELD_SIZE);
    sizzle = new Snake();
}
function draw() {
    background(50);
    sizzle.drawSegments();
    sizzle.move();
}
```

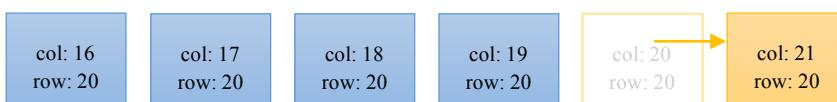
Wenn wir das Spiel jetzt neu starten sehen wir das der Schlangenkopf davon düst.

Das **Problem Nr. 2** ist auch sehr einfach zu lösen. Wir sollten p5 sagen wie schnell die **draw()** Funktion ausgeführt werden sollte. Dafür gibt es den Befehl **frameRate()**. Wie setze die Framerate auf 10 Frames (Bilder) pro Sekunde in dem wir den Befehl in **setup()** einfügen.

```
function setup() {
    createCanvas(COLS*FIELD_SIZE, ROWS*FIELD_SIZE);
    sizzle = new Snake();
    frameRate(10);
}
```

Schon viel besser!

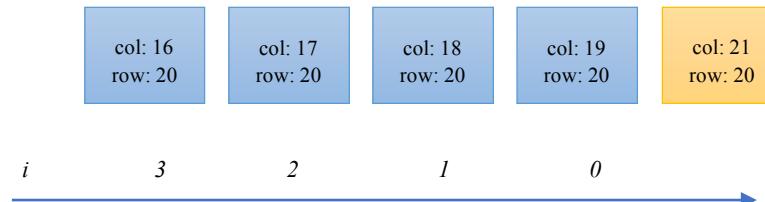
Das **Problem Nr. 3** wird einiges komplizierter. Wir müssen uns überlegen wie wir die Schwanz Segmente am besten zum Kopf aufschliessen lassen. Natürlich sollten wir dazu eine Schleife benutzen und ein Segment nach dem anderen abarbeiten. Wir sollten uns das einmal in einer Grafik anschauen.



Da der Kopf der Schlange schon um eine Position vorgerückt ist, ist seine alte Position (**col: 20, row: 20**) bereits verloren und wir wissen nicht wohin das nachfolgende Segment ziehen soll.

Es gibt bestimmt mehrere Lösungen für dieses Problem. Wir könnten zum Beispiel versuchen alle Positionen vor dem Vorrücken in separaten Variablen zwischen zu speichern.

Wir werden es aber etwas anders und eleganter tun. Wir können nämlich unsere Schleife so programmieren, dass wir das tail Array nicht von vorne nach hinten (von Index 0 bis ...) sondern von hinten nach vorne abarbeiten. Wenn wir mit dem letzten Segment beginnen, wissen wir auf welche Position wir es verschieben müssen – nämlich auf die Position des Elementes vor ihm.



```
move() {  
    for (let i=this.tail.length-1; i>0; i--) {  
        this.tail[i].col = this.tail[i-1].col;  
        this.tail[i].row = this.tail[i-1].row;  
    }  
  
    if (this.direction === 'RIGHT') {  
        this.head.col++;  
    }  
}
```

Wir sollten uns diese Schleife auch im Detail anschauen um sie zu verstehen.

Wieder leitet das Schlüsselwort **for** die Schleife ein und wieder gibt es drei Teile innerhalb der runden Klammern:

let i=this.tail.length-1;	Wir definieren eine Variabel mit dem Namen i , geben ihr aber nicht den Wert 0 wie in der anderen Schleife, sondern den Index des letzten Array Elementes. Wir benutzen dazu das lenght Attribut des Arrays, dieses ist aber um 1 zu gross, da der Index ja bei 0 beginnt. Deshalb müssen wir length mit 1 subtrahieren.
i > 0;	Wir vergleichen i mit 0. Die Schleife wird so lange wiederholt wie i grösser ist als 0.
i--	Vermindert den Wert von i um 1 nach jedem durchlaufen der Schleife.

Der Code in den geschwungenen Klammern sollten wir auch noch anschauen:

```
this.tail[i].col = this.tail[i-1].col;
```

Die Variabel **i** zeigt ja in unserer Schleife auf das aktuelle Element im tail Array: **this.tail[i]**

Wir wollen ja die Position des aktuellen Elements mit der Position des Elements vorne dran überschreiben. Wenn wir **i** um 1 reduzieren haben wir auch Zugriff auf dieses Element – deshalb steht in den zweiten eckigen Klammern **i-1** – wir holen uns damit den Wert **col** vom Segment vor dem aktuellen Segment.

Vielleicht hast Du bemerkt, dass unsere Schleife nicht alle Elemente im Array abarbeitet. Die Schleife wird nämlich nur solange ausgeführt wie **i** grösser als 0 ist. Das Segment im Element mit dem Index 0 wird also ausgelassen. Wir könnten die Schleife schon bis zum 0-ten Element laufen lassen, das würde aber zu einem Fehler führen. Wenn beim 0-ten Element **i=0** ist und wir versuchen mit **this.tail[i-1]** auf

das Element vorher zuzugreifen würde das Programm mit einem Fehler abbrechen, denn das Element mit dem Index -1 existiert nicht.

Zudem müssen wir das Segment an der vordersten Stelle auf die Position von **this.head** stellen. Das erledigen wir gleich nach der Schleife, noch bevor wir den Kopf bewegen.

```
move() {  
  
    for (let i=this.tail.length-1; i>0; i--) {  
        this.tail[i].col = this.tail[i-1].col;  
        this.tail[i].row = this.tail[i-1].row;  
    }  
    this.tail[0].col = this.head.col;  
    this.tail[0].row = this.head.row;  
  
    if (this.direction === 'RIGHT') {  
        this.head.col++;  
    }  
  
}
```

So, **Problem Nr. 3** ist auch gelöst. Wenn wir das Programm neu starten läuft sizzle nun endlich mit ihrem Schwanz los. Cool!

Im nächsten Kapitel sollten wir sizzle beibringen in verschiedene Richtungen zu laufen und auf unsere Pfeiltasten zu hören.

ÄNDERE DEINE RICHTUNG ...

Wir haben ja im Konstruktor der Snake Klasse das Attribut **direction** definiert und das auf RIGHT gesetzt. In der **Snake.move()** Methode haben wir implementiert was sizzle tun soll, wenn **direction** auf RIGHT gesetzt ist. Es ist nun an der Zeit dafür zu sorgen, dass unsere Schlange nicht nur nach rechts laufen kann.

Dazu implementieren wir zuerst eine neue Methode, welche es uns erlaubt sizzle mitzuteilen, dass sie ihre direction ändern soll. In der Snake Klasse schreiben wir die Methode **setDirection()**.

```
setDirection(newDirection) {  
    this.direction = newDirection;  
}
```

Wenn wir jetzt das Attribut direction von sizzle auf LEFT ändern wollen, können wir die Methode folgendermassen aufrufen:

```
sizzle.setDirection('LEFT');
```

Du wirst Dich jetzt vielleicht fragen warum tun wir das nicht einfach so:

```
sizzle.direction = 'LEFT';
```

Du hast recht. Das würde auch funktionieren. Aber in der Objekt Orientierten Programmierung ist es in der Regel so, dass man von ausserhalb von Objekten nicht direkt auf deren Attribute zugreift sondern dafür sogenannte get- und set- Methoden implementiert.

EVENT HANDLER

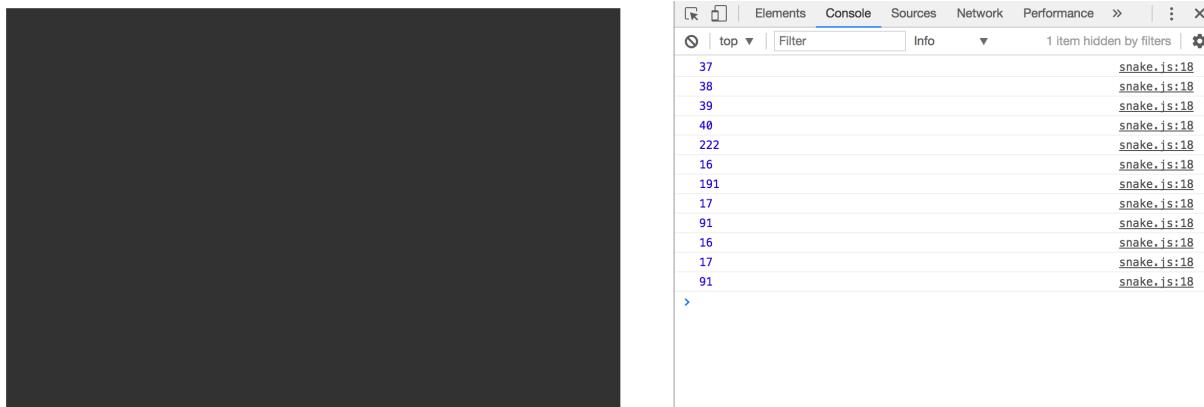
Die Snake Klasse hat nun die **setDirection()** Methode mit der wir das Attribut direction ändern können. Wir werden diese Methode in einem Event Handler benutzen.

Wenn immer Du am Computer etwas tust, zB die Maus bewegst oder eine Taste drückst wird das dem Programm mit einem Event (Ereignis) mitgeteilt. Momentan reagiert aber unser Programm noch auf keine dieser Events, denn wie das Programm auf Events reagieren soll müssen wir selber programmieren. Das tun wir jetzt.

Wir fügen nach der draw() Funktion eine neue Funktion mit dem Namen **keyPressed()** ein.

```
function keyPressed() {  
    console.log(keyCode);  
}
```

Die keyPressed() Funktion ist der Event Handler der bei jedem Drücken einer Tastatur Taste ausgeführt wird. Innerhalb von keyPressed gibt es die Variablen keyCode, welche wir mit **console.log()** auf unsere Browser Konsole ausgeben.



Wenn wir das Program neu starten und die Konsole öffnen sehen wir, dass die keyPressed Funktion bei jeder Taste komische Zahlen ausspuckt. Jede Taste hat ihren eigenen Zahlen Code und wir werden diese benutzen um Sizzle zu steuern.

Hier eine Zusammenfassung der für uns wichtigen Tasten Codes:

Taste	Code
Links	37
Oben	38
Rechts	39
Unten	40

Wir implementieren also in unserem keyPressed Event Handler einige if – Blöcke in denen wir prüfen welchen Code die Variablen keyCode hat.

```
function keyPressed() {  
  
    if (keyCode === 37) {  
        sizzle.setDirection('LEFT');  
    }  
    if (keyCode === 38) {  
        sizzle.setDirection('UP');  
    }  
    if (keyCode === 39) {  
        sizzle.setDirection('RIGHT');  
    }  
    if (keyCode === 40) {  
        sizzle.setDirection('DOWN');  
    }  
}
```

Wenn wir jetzt das Programm neu starten werden wir beobachten, dass Sizzle ein den Tasten Links, Oben und Unten stehen bleibt und bei Rechts wieder losläuft.

Wenn wir uns die move() Methode der Snake Klasse anschauen ist das auch genau was wir dort bisher programmiert haben. Wir haben erst implementiert wie sizzle auf direction === 'RIGHT' reagieren soll.

Wir ergänzen also noch die andren Richtungen in move():

```
move() {  
  
    for (let i=this.tail.length-1; i>0; i--) {  
        this.tail[i].col = this.tail[i-1].col;  
        this.tail[i].row = this.tail[i-1].row;  
    }  
    this.tail[0].col = this.head.col;  
    this.tail[0].row = this.head.row;  
  
    if (this.direction === 'RIGHT') {  
        this.head.col++;  
    }  
    if (this.direction === 'LEFT') {  
        this.head.col--;  
    }  
    if (this.direction === 'DOWN') {  
        this.head.row++;  
    }  
    if (this.direction === 'UP') {  
        this.head.row--;  
    }  
}
```

Endlich tut sizzle was wir wollen und kriecht in die richtige Richtung. Jetzt haben wir uns wirklich eine Belohnung verdient!

In den nächsten Kapiteln werden wir uns ums Futter & Wachsen kümmern.

FUTTER

Ziel des Spiels ist es ja, die Schlange Sizzle zum Futter zu steuern. Also kümmern wir uns als nächstes um das Futter.

Der Einfachheit halber werden wir unser Schlangenfutter als rotes Quadrat auf der Canvas zeichnen. Wir werden für das Futter eine eigene Klasse schreiben - die Klasse Food. Wenn wir uns überlegen was die Food Klasse alles können muss fällt uns auf, dass viel Programmcode identisch sein wird wie in der Segment Klasse. Wie bei einem Segment müssen wir die Koordinaten auf unserer Canvas berechnen und dort ein Quadrat zeichnen.

Wir könnten nun einfach den Code aus der Segment Klasse kopieren, aber es gibt eine elegantere Lösung. Wir werden nämlich die Segment Klasse erweitern. Man spricht hier in der Objekt Orientierten Programmierung auch von Vererbung. Unsere erweiterte Klasse Food wird den Code von Segment erben.

Wir beginnen mit der neuen Klasse Food so:

```
class Food extends Segment {  
}
```

Wie jede andere Klasse beschreiben wir die Klasse mit dem Befehl class aber erweitern (extend) Food von der Klasse Segment.

Tatsächlich könnten wir jetzt bereits Objekte von der Food Klasse erstellen und diese würden sich wie Segment Objekte verhalten. Alle Attribute und Methoden welche wir für Segment programmiert haben wurden in die Food Klasse vererbt. Wir können nun hingehen und die Food Klasse noch anpassen und gewisse Methoden anpassen oder neue dazu schreiben.

Beginnen wir mit dem Konstruktor:

```
class Food extends Segment {  
  
    constructor() {  
        super();  
  
        this.col = floor(random(0,COLS));  
        this.row = floor(random(0,ROWS));  
        this.color = color(200,50,50);  
    }  
}
```

Als erstes sehen wir, dass der Konstruktor der Klasse Food den Befehl **super()** enthält. Alle Konstruktoren von erweiterten Klassen müssen diesen Befehl aufweisen. Mit dem Befehl **super()** rufen wir nämlich den Konstruktor der Klasse Segment auf. Das stellt sicher, dass wir alle Attribute aus der Klasse Segment, welche wir ja in dessen Konstruktor definieren, auch in unserem Food Objekt definiert haben.

Unser Food Objekt hat also nach dem aufrufen von **super()** die Attribute **col**, **row**, **size** und **color**. Wir wollen diesen Attributen aber noch andere Werte vergeben und das geschieht in den folgenden Zeilen.

Sehen wir uns erst an was mit **col** und **row** passiert. Wir wollen dass unser Food Objekt eine Zufällige Position – und damit zufällige Werte für col und row erhält.

Der Befehl random() (zur Verfügung gestellt von p5.js) liefert eine Zufällige Kommazahl zwischen 0 und 1.

Du kannst das in der Browser Konsole ausprobieren indem Du den Befehl **random()** eingibst. Du wirst auch sehen, dass random() jedes Mal eine andere Zahl ausgibt.

Nun können wir mit einer Zahl zwischen 0 und 1 aber nicht viel anfangen. Unsere Zahl sollte einen Wert zwischen 0 und 40 haben – da unser Spielfeld 40 Felder hat.

Wir können der Funktion random() auch zwei Werte übergeben und so eine Zahl zwischen diesen Werten erhalten. Rufen wir also random(0, COLS) auf erhalten wir jedes Mal eine neue zufällige Zahl zwischen 0 und 40. (Wir haben ja COLS als Konstante mit dem Wert 40 definiert).

Nun wollen wir aber keine Kommazahlen, sondern ganze Zahlen. Auch dafür gibt es einen p5.js Befehl: **floor()**

Mit floor() können wir einer Kommazahl die Nachkommastellen abschneiden und erhalten den Wert vor dem Komma zurück. Rufen wir also floor(1.234) auf gibt floor() uns den Wert 1 zurück.

Füttern wir also floor() mit random(0,COLS) erhalten wir eine zufällige, ganze Zahl zwischen 0 und 40 und genau das geschieht im Konstruktor von Food mit col und row.

Damit unser Futter farblich anders aussieht als die Schlangen Segmente fügen wir dem Attribut color noch eine andere Farbe hinzu mit der Funktion **color(200,50,50)**. Im Segment Konstruktor haben for color() nur mit einer Zahl aufgerufen. Mit einer Zahl liefert color() eine graue Farbe zwischen Schwarz (0) und Weiss (255). Um richtige Farben zu generieren rufen wir color() mit drei Werten auf. Die drei Werte sind die Intensitäten für Rot, Grün und Blau. Die Mischung dieser drei Farben ist die Farbe welche color() dann liefert. Mit color(200,50,50) bekommt unser Food Objekt eine rote Farbe.

Um von unserer Food Klasse ein Objekt zu erstellen und auf der Canvas zu zeichnen müssen wir noch einige Zeilen Code schreiben:

Als erstes müssen wir eine Variabel erstellen in welcher das Food Objekt gespeichert wird. Wir definieren die Variabel food direkt unter der Variabel sizzle am Anfang des Programms:

```
const COLS=40;
const ROWS=40;
const FIELD_SIZE=15;
let sizzle;
let food;
```

In der Funktion setup() erstellen wir ein neues Food Objekt und speichern es in der Variabel food:

```
function setup() {
  createCanvas(COLS*FIELD_SIZE, ROWS*FIELD_SIZE);
  sizzle = new Snake();
  food = new Food();
  frameRate(10);
}
```

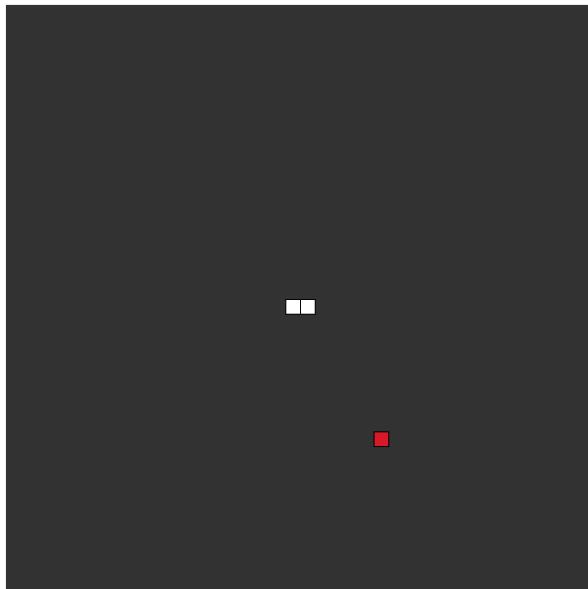
The screenshot shows a browser's developer tools console with the 'Console' tab selected. It displays a sequence of random number generation calls and their resulting values. The values are floating-point numbers between 0 and 1, with some being converted to integers by the floor() function. The output is as follows:

```
> random()
< 0.4273701576094573
> random()
< 0.08845615054117673
> random()
< 0.6925919274248695
> random()
< 0.0993227379105408
> random(0,COLS)
< 21.689026819120436
> random(0,COLS)
< 20.863751878402
> random(0,COLS)
< 4.431905302145749
> random(0,COLS)
< 12.122441783176221
> floor(random(0,COLS))
< 6
> floor(random(0,COLS))
< 18
> floor(random(0,COLS))
< 29
> floor(random(0,COLS))
< 2
>
```

Schlussendlich müssen wir dem Food Objekt noch den draw() Befehl aufrufen damit es sich zeichnet. Das tun wir wie bereits mit sizzle in der draw() Funktion:

```
function draw() {  
    background(50);  
    sizzle.drawSegments();  
    sizzle.move();  
    food.draw();  
}
```

Bei jedem Starten von unserem Programm sollten wir das Food Objekt an einer anderen zufälligen Position auf unserer Canvas sehen:



ESSEN

So, jetzt haben wir Futter auf der Canvas. Aber bevor Sizzle wachsen kann muss sie erst lernen zu Essen – und genau das werden wir als nächstes in diesem Kapitel programmieren.

Dazu braucht unsere Klasse Snake eine neue Methode **eat()** - diese wird nach jedem move() aufgerufen und prüft ob sich der Kopf der Schlange auf demselben Feld befindet wie das Futter.

Wenn wir uns das genau überlegen, werden wie die selbe Funktionalität auch später brauchen. Wir werden eine Funktion programmieren müssen um zu überprüfen ob sie die Schlange in den Schwanz beißt – also ob sich der Kopf der Schlange auf einem Feld des Schwanzes befindet. Da sich Futter, Kopf und alle Segmente des Schwanzes aus der Klasse Segment sind, macht es Sinn eine entsprechende Funktion in der Klasse Segment zu schreiben.

Segment soll also lernen zu überprüfen, ob es sich an derselben Position wie ein anderes Segment befindet. Dazu schreiben wir die Methode **isEqual()** in die Klasse Segment:

```
isEqual(otherSegment) {  
    if (this.col === otherSegment.col && this.row === otherSegment.row) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

In der isEqual() Methode ist eine if Anweisung wie wir sie bereits in anderen Teilen des Programms gesehen haben, nur, dass diese if Anweisung mehrere Vergleiche macht und sich zwischen zwei Programm Blöcken entscheidet.

Zuerst die Vergleiche in den runden Klammern ():

Zuerst werden die Werte von col verglichen – und zwar vom eigenen col (**this.col**) und vom col des anderen Segments (**otherSegment.col**) welches der Methode als Argument übergeben wird.

In einem zweiten Schritt werden auf dieselbe Wiese die Attribute row verglichen. Beide Vergleich werden mit **&&** verknüpft damit die if Anweisung „wahr“ entscheidet wenn beide Vergleiche „wahr“ sind.

Wenn beide Vergleiche wahr sind (col und row Werte stimmen überein), wird der Code Block in den ersten { } Klammern ausgeführt. Falls einer der col oder row Werte nicht überein stimmen wird der Code Block in den zweiten Klammern ausgeführt.

Bisher haben wir einige Methoden entwickelt, es ist aber das erste Mal, dass eine unserer Methoden etwas (einen Wert) zurückgibt. Deshalb lass uns kurz darauf eingehen. Eine Funktion oder Methode kann Werte beim Aufrufen entgegennehmen welche dann innerhalb des Codeblocks verarbeitet werden können. Es ist aber auch möglich einen Wert nach dem Abarbeiten der Funktion wieder zurückzugeben. Funktionen wie color(), random() oder floor() tun genau das. Unsere neue Methode isEqual() gibt einen Wert vom Typ Boolean zurück – das heisst der Rückgabewert kann „wahr“ oder „falsch“ sein. Den Befehl um etwas aus einer Methode oder Funktion zurückzugeben heisst **return**.

Ist das Segment also auf derselben Position wie das andere Segment gibt die Methode den Wert **true** (wahr) zurück. Falls die Positionen der beiden Segmenten nicht gleich ist bekommen wir von isEqual **false** (falsch).

So, nun können wir in Snake eine neue Methode schreiben welche prüft ob wir Futter gefunden haben. Wir erstellen dafür die Methode eat() in Snake. Darin prüfen wir mit der isEqual() Methode von Segment ob der Kopf der Schlange und das Futter Segment auf derselben Position sind:

```
eat() {  
    if ( this.head.isEqual(food) ) {  
        food = new Food();  
    }  
}
```

Da isEqual() „wahr“ oder „falsch“ zurück gibt, können wir den Aufruf von isEqual direkt in eine if() Anweisung schreiben. Gibt isEqual „falsch“ zurück passiert nichts. Gibt aber isEqual() „wahr“ zurück wird mit new Food() einfach ein neues Futter erstellt und dieses in der Variabel food gespeichert.

Das alte Food Objekt wird dabei gelöscht, weil es vom Neuen überschrieben wird. Da wir im Konstruktor von Food zufällige row und col Werte generieren werden neue Food Objekte immer an einer anderen Position erstellt.

Wir sollten unsere neue eat() Methode noch irgendwo aufrufen. Lass und eat() in der draw() Funktion direkt nach sizzle.move() platzieren.

```
function draw() {
    background(50);
    sizzle.drawSegments();
    sizzle.move();
    sizzle.eat();
    food.draw();
}
```

Wenn wir jetzt das Programm neu starten und Sizzle ein Futter essen lassen, sollte das Futter verschwinden und an einer anderen Position auftauchen.

Gratulation! Wieder eine wichtige Funktion des Spiels erledigt.

WACHSEN

Nun müssen wir noch implementieren, dass Sizzle nach dem Essen um ein Segment wächst. Das wird ein kurzes Kapitel, denn das ist sehr einfach.

Wir schreiben für die Klasse Snake eine neue Methode namens grow(). Wenn immer wir die Methode grow() aufrufen soll ein neues Segment Objekt erstellt werden und dieses an das Ende des tail[] Arrays hinzugefügt werden.

```
grow() {
    let segm = new Segment();
    this.tail.push(segm);
}
```

Alle JavaScript Arrays besitzen die Methode push() mit welcher man neue Elemente an das Ende des Arrays hinzufügen kann. Genau das tun wir hier mit der Variabel segm welches das neue Segment enthält.

Da Sizzle nach dem Essen wachsen soll rufen wir die grow() Methode in eat() auf:

```
eat() {
    if (this.head.isEqual(food)) {
        food = new Food();
        this.grow();
    }
}
```

Wenn Du das Programm neu startest wird Sizze nun nach jedem essen des Futters um ein Segment länger.

Super, wir machen grosse Schritte!

Übrigens, ist Dir aufgefallen, dass das neue Segment welches wir in grow() erzeugen und an das tail[] Array anhängen gar keine Position besitzt? Dem Konstruktor von Segment sollten wir ja die Variablen col und row übergeben, was wir in grow() nicht getan haben.

Weshalb weiss dann das neue Segment wo es gezeichnet werden soll?

Tatsächlich erstellen wir ein Segment ohne Position und fügen es mit tail.push() dem Array hinzu. Da wir aber beim nächsten Durchlauf von sizzle.move() mit dem letzten Segment in tail[] beginnen und so unserem positionslosen Segment die col und row Werte vom Segment davor geben ergeben sich deshalb keine Probleme im Programm.

SPIEL BEENDEN

Wir sind mit unserem Spiel Sizzle fast fertig. Aber wir müssen das Spiel noch beenden, wenn eine der folgenden Situationen eintrifft:

- Das head Segment befindet sich auf derselben Position wie eines der tail Segmente (die Schlange hat sich in den Schwanz gebissen).
- Das head Segment befindet sich ausserhalb der Canvas.

Wir schreiben dazu eine neue (und die letzte) Methode der Snake Klasse: checkEnd(), welche wir gleich in der draw() Funktion nach sizzle.move() aufrufen.

```
function draw() {  
    background(50);  
    sizzle.drawSegments();  
    sizzle.move();  
    sizzle.eat();  
    sizzle.checkEnd();  
    food.draw();  
}
```

Und die Methode checkEnd() soll so aussehen:

```
checkEnd() {  
    let end = false;  
  
    if (this.head.col > COLS) {  
        end = true;  
    }  
  
    if (this.head.col < 0) {  
        end = true;  
    }  
    if (this.head.row > ROWS) {  
        end = true;  
    }  
    if (this.head.row < 0) {  
        end = true;  
    }  
  
    for(let i=0; i<this.tail.length; i++) {  
        if ( this.tail[i].isEqual(this.head) ) {  
            end = true;  
        }  
    }  
  
    if ( end ) {  
        noLoop();  
    }  
}
```

In der ersten Zeile von checkEnd() erstellen wir eine Variabel mit dem Namen end und geben ihr gleich den Wert false.

Am Schluss von checkEnd() prüfen wir den Wert der Variabel end mit einer if() Anweisung und falls der Wert true ist lassen wir den Befehl noLoop() aufrufen. noLoop() ist ein p5 Befehl der p5 sagt wir wollen die Schleife (loop) welche die draw() Funktion immer wieder aufruft anhalten. Somit wird unser Programm sofort angehalten.

Dazwischen wird erst in vier if() Anweisungen überprüft, ob sich das die col und row Werte vom Schlangenkopf noch innerhalb der Canvas befinden. Falls nicht wird end auf true gesetzt.

Dann prüfen wir mit einer for Schleife jedes einzelne Segment vom tail[] Array mit unserer Methode isEqual() ob sich das Segment auf derselben Position wie das head Segment befindet. Falls das bei einem Segment der Fall ist wird auch end auf true gesetzt.

SCHLUSSWORT

Wenn Du tatsächlich bis hier durchgehalten hast und Dein Sizzle Programm funktioniert, dann ganz herzliche Gratulation – das war eine tolle Leistung!

Auf den nächsten Seiten findest Du das gesamte Programm nochmals abgedruckt, ergänzt mit Kommentaren, was die einzelnen Klassen und Methoden bedeuten.

Kommentare im Programmcode haben wir bisher noch gar keine geschrieben. Diese sind für einen Programmierer aber sehr wichtig. Denn es kommt bei jedem Programm vor, dass Du oder jemand anderes es nach einiger Zeit wieder anschaut – vielleicht um eine Verbesserung vorzunehmen oder einen Fehler zu beheben. Dann sind Kommentare sehr wichtig um zu verstehen wie es funktioniert und was sich der Programmierer dabei gedacht hat.

Kommentarzeilen werden in JavaScript mit einem // eingeleitet und beeinflussen den Ablauf vom Programm nicht.

Falls Du Lust hast das Programm zu verändern oder zu verbessern kannst Du das gerne tun. Hier einige Anregungen.

- Mach das Spielfeld grösser oder kleiner
- Verändere die Farben
- Lass das Programm schneller laufen je länger die Schlange wird
- Verteile mehr als ein Stück Futter

PROGRAMM CODE

INDEX.HTML

```
<html>
  <head>
    <script language="javascript" type="text/javascript"
src="https://p5js.org/assets/js/p5.min.js"></script>
    <script language="javascript" type="text/javascript" src="snake.js"></script>
  </head>
</html>
```

SNAKE.JS

```
// definiere die Anzahl Spielfelder in Kolonnen und Zeilen
const COLS=40;
const ROWS=40;
// definiere die Grösse eines Spielfeldes in Anzahl Pixel
const FIELD_SIZE=15;

// globale Variablen für Schlange und Futter
let sizzle;
let food;

// setup() wird von P5 einmal am Start des Programms ausgeführt
function setup() {

    // erstelle das Spielfeld
    createCanvas(COLS*FIELD_SIZE, ROWS*FIELD_SIZE);

    // erstelle die Objekte für Schlange und Futter
    sizzle = new Snake();
    food = new Food();

    // setze die Geschwindigkeit auf 10 Bilder pro Sekunde
    frameRate(10);
}

// draw() wird von P5 nach setup() in einer Schleife immer wieder aufgerufen
// und bildet den Spielablauf
function draw() {
    background(50);           // zeichne den Hintergrund der Canvas
    sizzle.drawSegments();    // zeichne alle Segmente der Schlange
    sizzle.move();            // berechne die nächste Position der Schlange
    sizzle.eat();             // prüfe ob die Schlange Futter gefunden hat
    sizzle.checkEnd();        // prüfe ob das Spiel geendet werden soll
    food.draw();              // zeichne das Futter
}

// Event Handler für Keyboard Tasten
// wird ausgeführt wenn eine Keyboard Taste gedrückt wird und steuert die
// Schlange in eine entsprechende Richtung
function keyPressed() {

    if (keyCode === 37) {
        sizzle.setDirection("LEFT");
    }

    if (keyCode==38) {
        sizzle.setDirection("UP");
    }

    if (keyCode==39) {
        sizzle.setDirection("RIGHT");
    }
    if (keyCode==40) {
        sizzle.setDirection("DOWN");
    }
}

// Klasse Snake
class Snake {

    // Konstruktor - wird ausgeführt wenn ein neues Objekt erstellt wird
    constructor() {
        this.direction = "RIGHT"; // Schlange soll in Richtung rechts los laufen
        this.head = new Segment(COLS/2, ROWS/2); // neues Segment für den Schlangenkopf in
der Mitte der Canvas
        this.tail = []; // ein leeres Array für die Schwanz Segmente

        // erstelle ein neues Segment links vom Kopf und füge es dem Array hinzu
        let ts1 = new Segment(this.head.col-1, this.head.row);
        this.tail.push(ts1);
    }

    // zeichnet alle Segmente der Schlange
    drawSegments() {

```

```
    this.head.draw();

    for(let i=0; i<this.tail.length; i++) {
        this.tail[i].draw();
    }
}

// berechnet die neue Position aller Segmente
move() {

    // neue Position für alle Schwanz Segmente (ausser das vorderste Segment)
    for(let i=this.tail.length-1; i>0; i--) {
        // aktuelles Segment bekommt Position vom Segment davor
        this.tail[i].col = this.tail[i-1].col;
        this.tail[i].row = this.tail[i-1].row;
    }

    // neue Position für das vorderste Segment - bekommt die Position vom Kopf
    this.tail[0].col = this.head.col;
    this.tail[0].row = this.head.row;

    // neue Position vom Kopf abhängig vom der gesetzten Richtung
    if (this.direction === "RIGHT") {
        this.head.col++;
    }

    if (this.direction === "LEFT") {
        this.head.col--;
    }

    if (this.direction === "DOWN") {
        this.head.row++;
    }

    if (this.direction === "UP") {
        this.head.row--;
    }
}

// setze die neue Richtung
setDirection(newDirection) {
    this.direction = newDirection;
}

// prüfe ob wir Futter gefunden haben
eat() {
    if ( this.head.isEqual(food) ) {
        // Futter gefunden! neues Futter erstellen und Schlange wachsen lassen
        food = new Food();
        this.grow();
    }
}

// erstellt ein neues Segment und fügt dieses dem Array tail hinzu
grow() {
    let segm = new Segment();
    this.tail.push( segm );
}

// überprüfe ob das Spiel beendet wird
checkEnd() {
    let end = false;

    // ist Schlangenkopf rechts aus der Canvas?
    if (this.head.col > COLS) {
        end = true;
    }

    // ist Schlangenkopf links aus der Canvas?
    if (this.head.col < 0) {
        end = true;
    }

    // ist Schlangenkopf unten aus der Canvas?
    if (this.head.row > ROWS) {
        end = true;
    }
}
```

```
// ist Schlangenkopf oben aus der Canvas?  
if (this.head.row < 0) {  
    end = true;  
}  
  
// ist Schlangenkopf auf einem Schwanzsegment?  
for(let i=0; i<this.tail.length; i++) {  
    if ( this.tail[i].isEqual(this.head) ) {  
        end = true;  
    }  
}  
  
if ( end ) {  
    // eines der oben geprüften Situationen ist eingetreten,  
    // stoppe die Loop und somit das Spiel.  
    noLoop();  
}  
}  
  
}  
  
// Klasse Segment  
class Segment {  
  
    // Konstruktor - wird ausgeführt wenn ein neues Objekt erstellt wird  
    constructor(col, row) {  
        this.col = col;  
        this.row = row;  
        this.size = FIELD_SIZE;  
        this.color = color(255);  
    }  
  
    // zeichne ein Quadrat an der Position von col und row  
    draw() {  
        fill(this.color);  
        rect(this.col*this.size, this.row*this.size, this.size, this.size);  
    }  
  
    // prüfe ob die Position mit der Position von einem anderen Segment übereinstimmt  
    isEqual(otherSegment) {  
  
        if (this.col === otherSegment.col && this.row === otherSegment.row) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}  
  
// Klasse Food welche Segment erweitert  
class Food extends Segment {  
  
    // Konstruktor - wird ausgeführt wenn ein neues Objekt erstellt wird  
    constructor() {  
        super(); // führt den Konstruktor der Super Klass (Segment) aus  
  
        // setze col und row auf eine zufällige Position  
        this.col = floor(random(0,COLS));  
        this.row = floor(random(0,ROWS));  
  
        this.color = color(200,50,50); // setze die Farbe auf Rot  
    }  
}
```