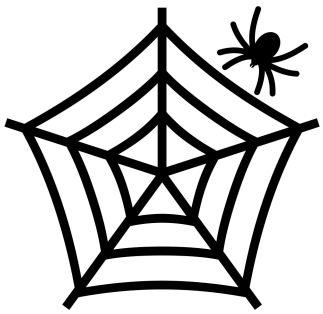
JavaScript

für Einsteiger



Eine Einführung in JavaScript
auf Basis von ECMAScript 6
mit der Bibliothek JSpider

für den Informatikunterricht
im Differenzierungsbereich

Version 1.1.1

Copyright:
Marco Haase
Viktoria-Gymnasium
45138 Essen

www.viktorianer.de

Inhalt

19 - Quax: Datenstruktur & Layout

In den bisherigen 18 Lektionen wurden die technischen Grundlagen gelegt, um mit JavaScript und der JSpider-Bibliothek auch (kleine) Computerspiele zu entwickeln. Allerdings: Diese „technischen Grundlagen“ sind zwar *notwendig*, um ein Spiel programmieren zu können, allerdings *nicht hinreichend*. Ähnlich wie Farbe, Pinsel und Leinwand zwar *notwendig* sind für ein großartiges Gemälde, aber eben *nicht hinreichend*.

In den letzten vier Lektionen werden wir Schritt für Schritt gemeinsam ein Computerspiel entwickeln. Ziel ist es nicht, Teile des Codes in eigene Programme einfach zu übernehmen, sondern aus dem Entwicklungsprozess und den Überlegungen auf dem Weg zum fertigen Spiel zu lernen, wie man dabei vorgehen kann.

Das Spiel heißt „**Quax**“. Man braucht dafür ein Kartenspiel mit 30 verschiedenen Karten (je 10 Karten in den Farben rot, grün und blau, die mit den Ziffern 0 bis 9 beschriftet sind) und zwei Spieler. Einen Computer braucht man (eigentlich) nicht. Die **Spielanleitung**:

> Alle Karten werden gemischt, danach bekommt jeder Spieler verdeckt vier Karten (sein so genanntes „**Blatt**“), die er auf die Hand nimmt und sich ansieht; der Gegner sollte sie nicht sehen. Die übrigen Karten werden als verdeckter **Ziehstapel** an die Seite gelegt.
>
> Das Spiel beginnt damit, dass ein Spieler eine seiner Karten offen hinlegt (Startspieler ist der jüngere der beiden). Der gegnerische Spieler sieht sich diese Karte an und legt dann seinerseits eine seiner vier Karten aus.
>
> Ziel ist es, diese beiden Karten (den so genannten „**Stich**“) für sich zu gewinnen. Den Stich bekommt derjenige Spieler, der die höherwertige Karte gelegt hat. Der Wert einer Karte entspricht dabei der aufgedruckten Ziffer. Bei Karten mit gleicher Ziffer ist die rote Karte mehr wert als die grüne, die grüne mehr als die blaue.
>
> Der Spieler, der den Stich gewinnt, legt die beiden Karten neben sich auf seinen **Ablagestapel**: Der Stich gehört ihm.
>
> Nun zieht jeder Spieler eine Karte vom Ziehstapel nach, so dass jeder wieder vier Karten auf der Hand hat. Damit ist eine Spielrunde beendet.
>
> In der nächsten Spielrunde legt derjenige Spieler zuerst eine Karte aus, der den vorherigen Stich gewonnen hat.
>
> Das Spiel ist zu Ende, wenn fünf Spielrunden gespielt wurden. Nun addiert jeder Spieler die Werte aller Karten auf seinem Ablagestapel – das ist die Gesamtpunktzahl. Das Spiel gewonnen hat der Spieler mit der höchsten Gesamtpunktzahl.
>
> Haben beide Spieler dieselbe Gesamtpunktzahl, dann gibt es keinen Gewinner – das Spiel geht unentschieden aus.
>
> Wird mehr als ein Spiel gespielt, dann ist der Verlierer des vorherigen Spiels (falls es einen gab) jeweils der Startspieler des neuen Spiels; ansonsten eben der jüngere Spieler.

Für eine Umsetzung als Computerspiel ist zunächst die **Rolle des Computers** zu klären:
Variante 1: Der Computer stellt „nur“ die Plattform zur Verfügung, es spielt „Mensch gegen Mensch“.
Vorteil: Es muss keine „künstliche Intelligenz“ (KI) programmiert werden.
Nachteil: Es müssen zwei Spieler abwechselnd am selben Gerät spielen. Man kann nicht allein spielen.
Variante 2: Der Computer übernimmt selbst die Rolle eines Spielers, es spielt „Mensch gegen Maschine“.
Vorteil: Man kann alleine spielen.
Nachteil: Der Computer muss „schlau gemacht“ werden und das ist – je nach Spiel – gar nicht so einfach.

Wir werden Variante 2 umsetzen, weil man dabei mehr lernen kann und weil das Spielen mehr Spaß macht. Es spielt also „**Mensch gegen Maschine**“.

Häufig unterschätzt, aber im Grunde wichtiger als das Layout ist eine **passende Datenstruktur**. Meist gibt es nicht nur *eine* gute Lösung, allerdings viele schlechte. Ein paar Überlegungen für unser Spiel:

Benötigt wird ein Kartenspiel aus 30 Karten. Jede **Karte** hat eine Farbe und einen Wert. Da die Werte nur einstellige Zahlen sind und die Farben unterschiedliche Anfangsbuchstaben haben, kann man jede Karte durch eine **Zeichenkette** der Form **"5G"** (= grüne 5) oder **"3R"** (= rote 3) darstellen. Diese Darstellung hat (z.B. gegenüber "G5" oder "R3" oder ganz anderen Varianten) einen entscheidenden Vorteil: Ein Größenvergleich zweier solcher Zeichenketten liefert direkt das korrekte Ergebnis für den Wertvergleich der beiden Karten! Und das wird benötigt, denn es muss ja (vom Computer) entschieden werden, welche von zwei (oder mehr) Karten die höchste ist!

Erklärung: Der Vergleich zweier Zeichenketten funktioniert wie beim Wörterbuch: Es wird buchstabenweise verglichen: "7" ist größer als "3" und "R" ist größer als "G". Da „rot“ mehr als „grün“ zählt und „grün“ mehr als „blau“ (siehe Spielanleitung) und dies der Reihenfolge der Buchstaben R, G und B im Alphabet entspricht, ergibt der alphabetische Vergleich immer den korrekten Wertevergleich.

Diese String-Repräsentation der Karten wird für die Spiellogik verwendet, nicht jedoch für die Darstellung auf dem Bildschirm. Da Karten bewegt werden müssen, liegt es nahe, dafür **Spider-Objekte** zu benutzen.

Im Spiel haben wir es mit verschiedenen „Kartensammlungen“ zu tun: Ziehstapel, Blatt und Ablagehaufen der beiden Spieler. Es bietet sich an, diese Sammlungen jeweils als **Array von Karten** aufzufassen. Je nach Art der Kartensammlung müssen Karten weggenommen oder hinzugefügt werden – entsprechende Methoden dafür kennen wir (↗ Lektion 11).

Wir können nun die **Datenstruktur** anlegen. Zunächst der **Ziehstapel**:

```
let ziehstapel = [];
for (let z=0; z<=9; z+=1) {
    for (let f of "RGB") {
        ziehstapel.push(z+f)
    }
}
ziehstapel.shuffle(); // Neu: Karten mischen!
```

Dann benötigen wir Arrays für die verschiedenen **Kartensammlungen**:

```
let spielerBlatt = [];  // Kartenwerte z.B. "R7"
let spielerKarte = [];  // Karten-Objekte
let spielerAblage = []; // Kartenwerte auf Ablage
let spielerWahl;        // Karte Nr. 0,1,2 oder 3
let computerBlatt = [];
let computerKarte = [];
let computerAblage = []
let computerWahl;
```

Und dann noch einige weitere **globale Variablen** für den Spielablauf:

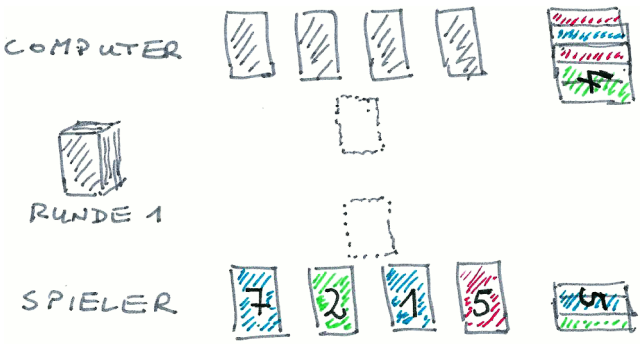
```
let runde = 1; // der Rundenzähler
let start = "Spieler"; // wer die Runde beginnt

let farbe = new Map(); // RGB → Kartenfarben
farbe.set("R","salmon");
farbe.set("G","palegreen");
farbe.set("B","skyblue");
```

Aufgaben

1. Beginne mit dem Programm **Quax**, indem du die obigen Teile übernimmst. Sieh dir den Inhalt des Ziehstapels mit writeln() an – einmal *vor* und einmal *nach* dem Mischen.

Bevor wir mit der Entwicklung der Spieloberfläche beginnen, noch einmal der Hinweis: Die **Entwicklung einer tragfähigen Datenstruktur** ist nicht einfach und erfordert viele Überlegungen. Dennoch ist dieser Schritt **unverzichtbar** und häufig entscheidend dafür, ob man am Ende den eigenen Quelltext noch versteht oder überhaupt ein lauffähiges Programm entsteht!

Als nächstes beschäftigen wir uns mit dem **Entwurf der Spieloberfläche**, und zwar zunächst einmal mit Papier und Farbstiften. Mein Entwurf sieht z.B. so aus:



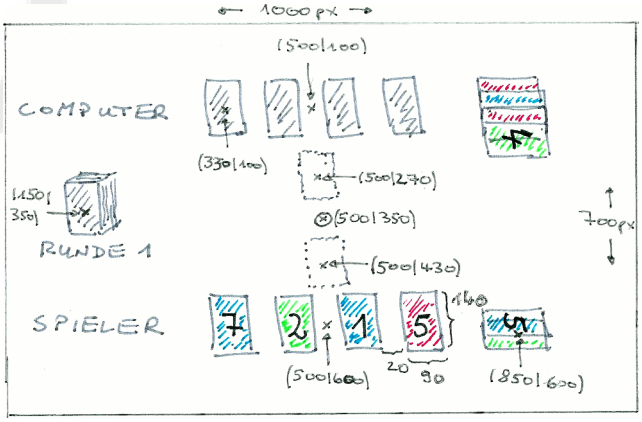
Es handelt sich um einen *Entwurf*, an dem man sich orientieren kann. Stellt sich bei der Entwicklung des Programms heraus (und das ist der Normalfall), dass Dies oder Jenes nicht bedacht wurde oder anders besser ist, dann ändert man den Entwurf entsprechend.

Der nächste Schritt besteht darin, den Entwurf in eine passende Spider-World zu übertragen, d.h. Maße, Positionen und Abstände für die einzelnen Elemente festzulegen (und zu notieren).

Ein sinnvoller Ausgangspunkt für diese Planungen sind die Abmessungen des gesamten Spielfeldes. Es soll einerseits möglichst groß sein, um speziell auf kleinen Displays noch spielbar zu sein, und andererseits nicht so groß, dass Bildschirme mit niedriger Auflösung (z.B. 1366x768 oder 1024x768) das Spielfeld nicht komplett darstellen können. Eine **Feldgröße von 1000x700 px** sollte ein guter Kompromiss sein.

Standard-**Spielkarten** haben eine Größe von 59x91 mm. Man könnte das 1:1 in px übertragen, aber das wäre unnötig klein. Bei einer Vergrößerung sollte man darauf achten, dass das Seitenverhältnis erhalten bleibt, damit das Kartenformat aussieht wie wir es gewohnt sind. Nach einigen Größenexperimenten zeigt sich, dass **90x140 px** ein gutes Format mit einem Seitenverhältnis von annähernd 59:91 ist.

Bei einer gut verteilten Anordnung der anderen Komponenten ergeben sich deren Positionen von selbst – mit etwas Rechnen und etwas Probieren. Einige dieser Angaben sollte man im Entwurf notieren – das hilft später bei der Orientierung. Etwa so:



Aufgaben

2. Fertige einen eigenen (farbigen, handgezeichneten) Entwurf für die Spieloberfläche von Quax an (Din A4 quer). Trage ausgewählte Maße und Punkte dort ein.

20 - Quax: Spielstart

Die Ausgangslage vor Spielstart soll nun in die grafische **Darstellung der Spieloberfläche** umgesetzt werden. Wir brauchen das Spielfeld, einen Ziehstapel und die im Entwurf vorgesehene Beschriftung:

```
let feld = new World(1000,700);
let stift = new Spider(feld);
stift.speed = 100;
stift.fontsize = 36;
stift.textalign = "center";
// Beschriftungen
stift.jumpTo(150,100,"noturn");
stift.print("Computer");
stift.jumpTo(150,600,"noturn");
stift.print("Spieler");
zeigeRunde(); // Zähler aktualisieren und anzeigen
// Ziehstapel darstellen
let karte = neueKarte();
karte.stamp();

// ... hier folgt der Rest des Hauptprogramms

feld.animate(); // nicht vergessen!
```

Zum Einsatz kommen hier (und im weiteren Verlauf noch öfter) **eigene Funktionen**, die zunächst wie eine **Black-Box** eingesetzt werden: *Wie* diese Funktionen genau implementiert werden, bleibt zunächst noch offen, aber *was* die Funktionen leisten (sollen), ist klar:
zeigeRunde() erhöht den Rundenzähler um 1 und aktualisiert die Rundenzähleranzeige im Feld.
neueKarte() erzeugt ein Spider-Objekt in Kartengestalt, das die Rückseite der Karte zeigt und dort positioniert ist, wo der Ziehstapel ist.

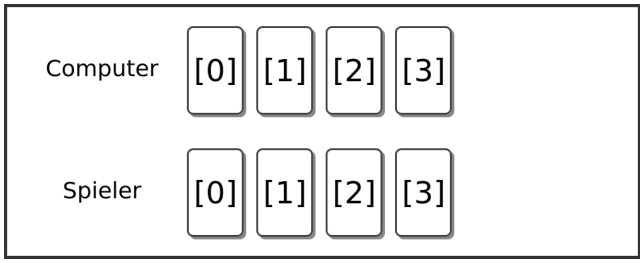
Diese Funktionen müssen später noch entwickelt werden. Vorerst genügt es aber zu wissen, was die Funktionen leisten sollen und dass man in der Lage ist, solche Funktionen zu erstellen.

Nun werden die Karten verteilt: Je vier Karten für Computer und Spieler:

```
for (let nr=0; nr<=3; nr+=1) {
    zieheKarte(neueKarte(),"Computer",nr);
    zieheKarte(neueKarte(),"Spieler",nr);
}
```

Erneut kommen Black-Box-Funktionen zum Einsatz:
zieheKarte() sorgt dafür, dass die oberste Karte des Ziehstapels an eine bestimmte Position im Blatt des Computers bzw. Spielers verschoben wird.
Das *erste Argument* ist das dafür verwendete Karten-Objekt. Zu Spielbeginn müssen die Karten einmalig erzeugt werden, im weiteren Verlauf werden diese Objekte wiederverwendet. Das *zweite Argument* gibt die Zielperson an und das *dritte* die Nummer der Karte im Blatt der Zielperson, die zugleich dem Index entspricht, der dieser Karte in den Arrays `spielerBlatt[]` und `spielerKarte[]` entspricht.

Diese Nummern bzw. Indizes sind dem Blatt der beiden Spieler wie folgt zugeordnet:



Die Karten sind nun verteilt, das Spiel kann beginnen. Aber wer fängt an? Wir legen fest: Der Spieler ist immer jünger als der Computer und beginnt das Spiel. Damit der Spieler spielen kann, muss er eine seiner Karten auswählen können. Wie soll er dies tun?

Intuitiv wäre wahrscheinlich „drag & drop“. Das erlaubt allerdings auch Fehlbedienungen (Karte wird z.B. auf den Ziehstapel geschoben), die im Event-Handler geprüft und unterbunden werden müssen. Wir wählen darum einen aus Programmierersicht einfacheren Weg: Per Doppelklick auf die gewünschte Karte spielt der Spieler seine Karte aus. Die Kartenobjekte müssen also noch in den Lauschzustand versetzt werden. Da das im Programm noch an anderen Stellen nötig ist, verwenden wir auch dafür eine eigene Funktion:

```
lauschKarten(true); // Spielerblatt "lauscht"
```

Je nach Argument – `true` oder `false` – wird das Lauschen der Karten ein- oder abgeschaltet. Auch das Abschalten des Lauschens wird benötigt, damit der Spieler z.B. nicht zwei Karten auswählt oder eine Karte auswählt, wenn der Computer gerade am Zug ist. Die Umsetzung dieser Funktion könnte so aussehen:

```
const lauschKarten = function (handler) {
    if (handler) {handler = spielerZug;}
    for (let nr=0; nr<=3; nr+=1) {
        spielerKarte[nr].listen("dblclick",handler);
    }
}
```

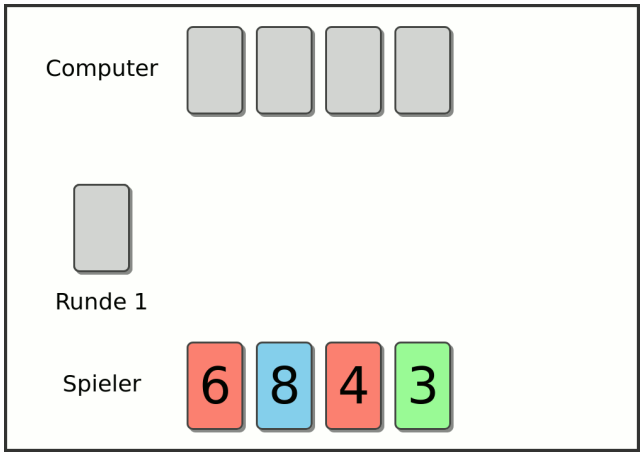
In der ersten Zeile wird geprüft, ob der Wert von `handler` `true` ist – dann soll das Lauschen eingeschaltet werden: `handler` wird die Funktion zugewiesen, die als Event-Handler dienen soll (und bisher nur als Black-Box existiert: ***spielerZug***()). Beachte, dass nur der Funktions*name* (ohne die Funktions*klammern*) zugewiesen wird – ansonsten würde die Funktion sofort ausgeführt und nicht erst bei Doppelklick.

In der Schleife wird allen vier Karten-Objekten des Spielers, die im Array `spielerKarte[]` abgelegt sind, nun das Lauschen verordnet. Oder – falls `handler` den Wert `false` hat – das Lauschen abgestellt.

Aufgaben

1. Ergänze das Quax-Programm um alle Teile dieser Lektion.

Der Spieler sieht nun – nachdem die anderen Black-Box-Funktionen implementiert wurden! – etwa Folgendes und kann eine Karte per Doppelklick ausspielen:



Bevor wir uns überlegen, wie es danach weitergeht, hier zunächst die **Quelltexte der Funktionen**:

```
// Karte erzeugen und auf Ziehstapel positionieren
const neueKarte = function () {
    let karte = new Spider(feld);
    karte.image = "";
    karte.width = 90;
    karte.height = 140;
    karte.border = "solid 3px #444";
    karte.edges = "10px";
    karte.shadow = "4px 4px #888";
    karte.fontsize = 80;
    karte.color = "lightgray";
    karte.speed = 100;
    karte.jumpTo(150,350,"noturn"); // Ziehstapel
    karte.speed = 10; // sinnvolle Kartengeschw.
    karte.visible = true;
    return karte
}
```

```
// Karte vom Ziehstapel ins Blatt befördern
const zieheKarte = function (karte,spieler,nr) {
    let wert = ziehstapel.shift(); // "ziehen"
    karte.label = "";
    karte.color = "lightgray"; // Rückseite
    karte.visible = true;
    if (spieler === "Spieler") { // Spieler-Karte
        spielerBlatt[nr] = wert;
        spielerKarte[nr] = karte;
        karte.moveTo(330+nr*110,600,"noturn");
        // Karte "umdrehen": Wert und Farbe zeigen
        karte.label = wert[0]; // Zahlenwert
        karte.color = farbe.get(wert[1]); // Farbe
        karte.nr = nr; // Neu: Karten-Nummer!
    } else { // Karte für Computer
        computerBlatt[nr] = wert;
        computerKarte[nr] = karte;
        karte.moveTo(330+nr*110,100,"noturn");
    }
}
```

Der Aufbau der Funktion neueKarte() sollte klar sein, ***zieheKarte***() ist jedoch erklärungsbedürftig:

Zunächst wird die oberste Karte des Ziehstapels aus dem Array entnommen – das gehört zur Programmlogik und hat keine *sichtbaren* Auswirkungen.
Karten, die vom Ziehstapel kommen, zeigen ihre Rückseite. Das tun *neue* Karten zwar automatisch, aber im weiteren Verlauf werden ausgespielte Karten-Objekte (unsichtbar) auf den Ziehstapel geschoben, um von dort als scheinbar „neue“ Karte an Spieler oder Computer verteilt zu werden. Und *diese* Karten-Objekte, die vorher auf den Ablagestapel verschoben wurden, zeigen ihre Vorderseite. Darum: Rückansicht festlegen.

Die gezogene Karte wird nun in das Blatt des Spielers bzw. des Computers an der gewünschten Stelle *nr* eingeordnet – und zwar einmal *logisch* mit dem Kartenwert im *spielerBlatt[]* und einmal *sichtbar* als Karten-Objekt in *spielerKarte[]*.

War es der Spieler, der eine Karte gezogen hat, dann muss die Kartenvorderseite sichtbar gemacht werden, sobald die Karte ihr Ziel im Spielerblatt erreicht hat. Außerdem benötigen die Karten-Objekte im Blatt des Spielers ein zusätzliches Attribut ***.nr***, in dem steht, welche Nummer (0…3) diese Karte hat. Da die Karte per Mausklick ausgewählt wird, kennt der Event-Handler über das *source*-Attribut des Event-Objekts zwar das Karten-Objekt selbst, nicht aber die Nummer im Blatt des Spielers – und die wird benötigt.

```
// Rundenzähleranzeige aktualisieren
const zeigeRunde = function () {
    runde += 1;
    feld.clear(60,440,240,500);
    stift.jumpTo(150,470,"noturn");
    stift.print("Runde "+runde);
}
```

```
// Event-Handler für Karten-Doppelklick
const spielerZug = function (event) {
    lauschKarten(false); // Lauschen abstellen!
    let karte = event.source; // ausgewählte Karte
    karte.moveTo(500,430,"noturn"); // ausspielen
    spielerWahl = karte.nr; // Auswahl merken!
    if (start === "Spieler") {
        computerZug();
    } else {
        stichAuswertung();
    }
}
```

Wichtig ist die Fallunterscheidung zum Schluss: Falls der Spieler die Runde begonnen hat, dann ist jetzt der Computer dran. Hat der Spieler *nicht* begonnen, dann war der Computer bereits dran – dann muss der Stich nun ausgewertet werden.

21 - Quax: Spielablauf

Als nächstes ist der **Computer am Zug** – die Funktion ***computerZug()***, bisher nur als Black-Box vorhanden, wird ausgeführt. Anders als der Spieler kann (und muss) der Computer nicht auf eine Karte klicken, sondern schiebt sie einfach nach vorne in die Mitte:

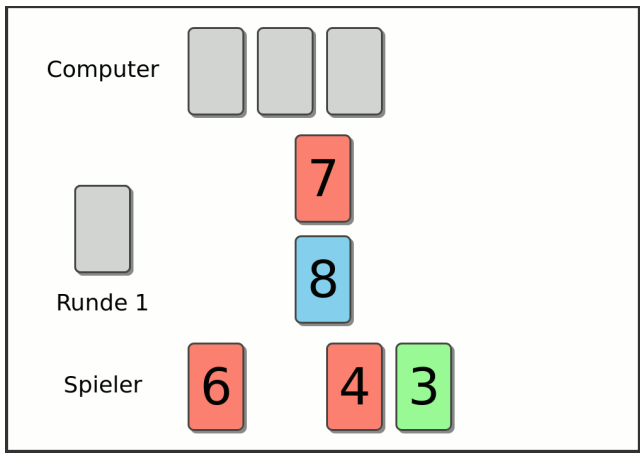
```
// Spielzug des Computers
const computerZug = function () {
    computerWahl = computerEntscheidung();
    // Karte ausspielen und "umdrehen"
    let wert = computerBlatt[computerWahl];
    let karte = computerKarte[computerWahl];
    karte.label = wert[0]; // Zahlenwert
    karte.color = farbe.get(wert[1]); // Farbe
    karte.moveTo(500,270,"noturn"); // ausspielen
    if (start === "Computer") {
        lauschKarten(true);
    } else { // Spieler hat begonnen
        stichAuswertung();
    }
}
```

Der Unterschied zu *spielerZug*() liegt vor allem im Entscheidungsprozess: Dort ist es der Spieler, der entscheidet und klickt, nun muss der Computer selbst entscheiden und eine Karte auswählen. Das Ergebnis des Entscheidungsprozesses – bei Spieler und Computer – ist die Nummer der ausgewählten Karte. Diese Nummer liefert die Funktion ***computerEntscheidung***().

Damit Quax möglichst bald gespielt werden kann, stellen wir die Überlegungen und die Umsetzung einer Entscheidungslogik („Künstliche Intelligenz" – KI) noch zurück. Vorerst begnügen wir uns damit, dass der Computer einfach zufällig eine Karte auswählt:

```
// Entscheidungsprozess des Computers ("KI")
const computerEntscheidung = function () {
    return randomNumber(0,3);
}
```

Nun sind beide Karten ausgespielt, die Situation könnte also wie folgt aussehen:



Da der Spieler die Runde begonnen hat (d.h. es gilt *start==="Spieler"*) erfolgt nun die Auswertung: Wer bekommt den Stich und was geschieht, nachdem das geklärt ist? Hätte der Computer die Runde begonnen, dann wäre nun der Spieler am Zug: Die Karten des Spielers würden wieder in den Lauschzustand versetzt.

Nun also die **Auswertungsphase**. Zu entscheiden, wer den Stich bekommt, ist einfach – dank der von uns gewählten Repräsentation der Karten. Ausgespielt wurden "5G" und "8R". Ein einfacher Vergleich dieser beiden Zeichenketten (= Kartenwerte) führt zum richtigen Ergebnis: Der Spieler bekommt den Stich.

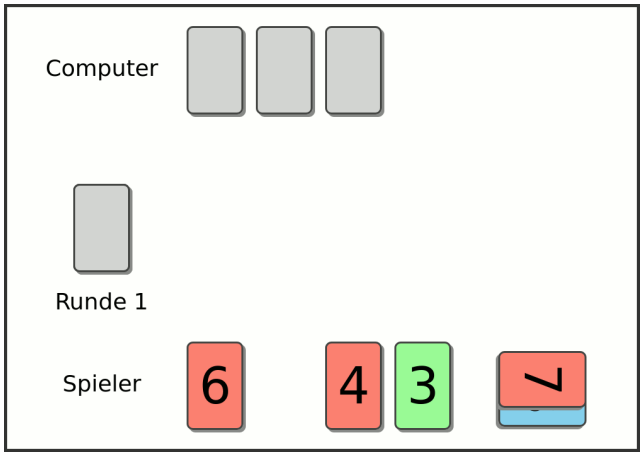
```
// Auswertung einer Spielrunde
const stichAuswertung = function() {
    stift.wait(500); // einen Moment Bedenkzeit
    let sw = spielerBlatt[spielerWahl]; // Wert
    let sk = spielerKarte[spielerWahl]; // Objekt
    let cw = computerBlatt[computerWahl];
    let ck = computerKarte[computerWahl];
    // Wer bekommt den Stich?
    if (sw > cw) { // Spieler gewinnt Stich
        start = "Spieler"; // für nächste Runde
        legeKarteAb(sk,850,620-spielerAblage.length*30);
        legeKarteAb(ck,850,590-spielerAblage.length*30);
        spielerAblage.push(sw,cw);
    } else { // Computer gewinnt Stich
        start = "Computer";
        legeKarteAb(ck,850,80+computerAblage.length*30);
        legeKarteAb(sk,850,110+computerAblage.length*30);
        computerAblage.push(sw,cw);
    }
}
```

Damit der Quelltext besser lesbar ist, werden lokale Hilfsvariablen für die Kartenwerte (sw, cw) und Karten-Objekte (sk, ck) eingeführt. Je nachdem, wer den Stich gewinnt, wird als Startspieler für die nächste Runde der Verlierer dieser Runde festgelegt und dann die beiden Karten des Stichs dem korrekten Ablagestapel zugeordnet – einmal logisch (Kartenwerte in das Ablage-Array) und einmal sichtbar. Letzteres übernimmt die Funktion ***legeKarteAb****(karte,x,y)*:

```
// Karte auf Ablagestapel (x|y) schieben
const legeKarteAb = function (karte,x,y) {
    karte.moveTo(x,y,"noturn");
    karte.turn(-90);
    karte.stamp();
    karte.visible = false;
    karte.turn(90);
}
```

BEACHTE: Da die Karten auf dem Ablagestapel gefächert sein sollen (siehe Layout-Entwurf), so dass man die Anzahl der Karten erkennen kann, ist der y-Wert der Ablageposition von der Anzahl der Karten abhängig, die vorher bereits auf dem Ablagestapel sind.

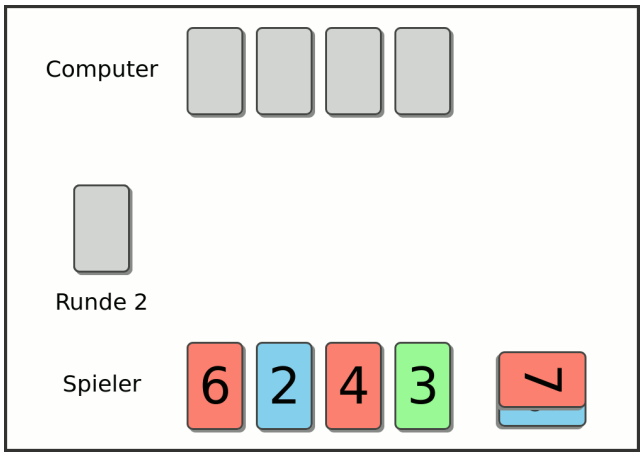
So, der Stich ist ausgewertet, die Karten wurden auf den Ablagestapel verschoben:



Wie es weitergeht, hängt davon ab, wie viele Runden bereits gespielt wurden, denn nach der 5. Runde ist Schluss. Andernfalls bekommen Spieler und Computer je eine neue Karte vom Ablagestapel. Dies muss in der Funktion ***stichAuswertung***() noch ergänzt werden:

```
if (runde === 5) { // Spielende -> Auswertung
    spielAuswertung();
} else { // Weiterspielen: Karten nachziehen
    sk.jumpTo(150,350,"noturn");
    zieheKarte(sk,"Spieler",spielerWahl);
    ck.jumpTo(150,350,"noturn");
    zieheKarte(ck,"Computer",computerWahl);
    zeigeRunde();
    if (start === "Spieler") {
        lauschKarten(true);
    } else {
        computerZug();
    }
}
```

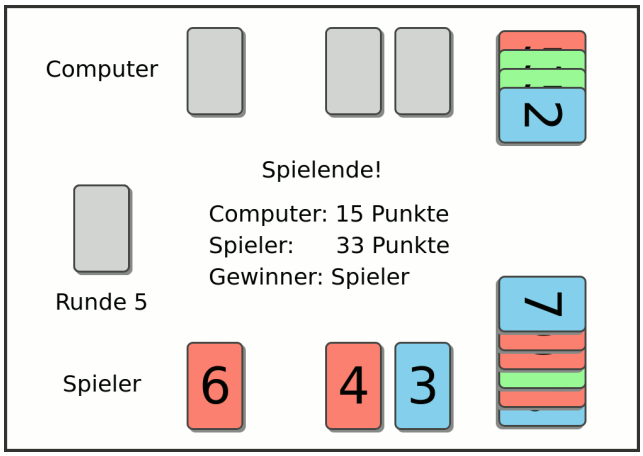
Die Funktion ***zieheKarte***() (↗ Lektion 20) erwartet u.a. ein Karten-Objekt, das sich oben auf dem Ziehstapel befindet. Die zwei abgelegten Karten (die auf dem Ablagestapel nur als Abdruck existieren) werden also zunächst (unsichtbar) dorthin versetzt und anschließend die Karte als neue Karte in das Blatt des Spielers bzw. Computers (an der leeren Stelle) eingeordnet. Danach wird der Rundenzähler aktualisiert:



Abschließend wird geprüft, wer die nächste Runde beginnt und – je nach dem – entweder das Spieler-Blatt in den Lauschzustand versetzt oder der Computer angewiesen, als erster eine Karte auszuspielen.

Damit ist der Spielablauf vollständig – Quax kann gespielt werden. Es fehlt nur noch die **Spielauswertung**: Die Werte der abgelegten Karten befinden sich in den beiden Ablage-Arrays, die Punktwerte müssen addiert und die Summen verglichen werden. Abschließend wird das Ergebnis in der Spielfeldmittel ausgegeben:

```
const spielAuswertung = function () {
    // Punktestand berechnen, Gewinner ermitteln
    let spielerPunkte = 0;
    let computerPunkte = 0;
    for (let wert of spielerAblage) {
        spielerPunkte += Number(wert[0]);
    }
    for (let wert of computerAblage) {
        computerPunkte += Number(wert[0]);
    }
    let gewinner = "Keiner"; // Grundannahme ...
    if (spielerPunkte < computerPunkte) {
        gewinner = "Computer";
    }
    if (spielerPunkte > computerPunkte) {
        gewinner = "Spieler";
    }
    // Auswertungstext anzeigen
    stift.jumpTo(500,260,"noturn");
    stift.print("Spielende!");
    stift.textalign = "left";
    stift.jumpTo(320,330,"noturn");
    stift.print("Computer: "+computerPunkte+" Punkte");
    stift.jumpTo(320,380,"noturn");
    stift.print("Spieler:    "+spielerPunkte+" Punkte");
    stift.jumpTo(320,430,"noturn");
    stift.print("Gewinner: "+gewinner);
}
```



Aufgaben

1. Ergänze das Quax-Programm um alle Teile dieser Lektion und spiele einige Runden Quax.

22 - Quax: KI und Optimierung

Quax funktioniert! – Warum also noch eine Lektion? Zur Erinnerung: Der Computer trifft seine Entscheidungen quasi „aus dem Bauch heraus“ und ist dadurch langfristig auf jeden Fall der unterlegene Spieler. Unser Ziel soll es sein, den Computer-Spieler genauso gut spielen zu lassen, wie einen (halbwegs begabten) menschlichen Spieler.

Theoretisch kennt der Computer natürlich die Karten des menschlichen Spielers, ebenso wie alle Karten des Ziehstapels und deren Reihenfolge – diese Informationen könnte der Computer (aus)nutzen, um sich Spielvorteile zu verschaffen. Das wäre jedoch unfair und wird natürlich *nicht* gemacht. Alle Spielentscheidungen trifft der Computer nur aufgrund der Kenntnis seiner eigenen Karten und der bereits abgelegten Karten – wie der menschliche Spieler auch.

Unter diesen Voraussetzungen wollen wir den Computer nun zu einem guten Quax-Spieler machen. Das geschieht in zwei Schritten:

Schritt 1: **Entscheidungsregeln formulieren**.
Der Computer braucht klare Regeln, nach denen er seine Entscheidungen trifft, am besten als „Wenn-Dann-Sätze“ formuliert: „Wenn dieser Fall eintritt, dann tue dies“ (aus Programmierersicht: „bedingte Anweisung“).
Wie kommt man nun zu solchen Regeln, und zwar am besten solchen, die den Computer zu einem guten Spieler machen? – Man muss sein eigenes Spielverhalten analysieren und möglichst klar und differenziert formulieren, bei welcher Spielsituation man wie entscheiden würde.

Bei Quax lassen sich drei Fälle unterscheiden und in Form von **Wenn-Dann-Regeln** formulieren:
Regel 1: ***Wenn*** man als erster am Zug ist, ***dann*** spielt man die Karte mit dem höchsten Wert aus.
Regel 2: **Wenn** man als zweiter am Zug ist und (mindestens) eine Karte hat, die höher ist als die ausgespielte Karte des Gegners, **dann** spielt man die niedrigste dieser höheren Karten aus, um den Stich zu gewinnen.
Regel 3: **Wenn** man als zweiter am Zug ist und keine Karte hat, die höher ist als die ausgespielte Karte des Gegners, **dann** spielt man seine niedrigste Karte aus, weil der Stich sowieso verloren ist.

Das sind noch keine optimalen Regeln, aber wenn der Computer nach diesen Regeln spielt, hat er zumindest eine realistische Gewinnchance. Eine mögliche Optimierung wäre durch folgende **Regel** möglich:
Wenn man die vom Gegner ausgespielte Karte überbieten kann und es die letzte Runde ist, **dann** legt man seine höchste Karte (weil man sie für nichts anderes mehr braucht und die Punktsumme so maximiet wird).

Schritt 2: **Entscheidungslogik implementieren**.
Nun geht es darum, die Entscheidungsregeln in die Sprache des Computer (d.h. für uns: JavaScript) zu übertragen (zu „implementieren“).
Da die Kartenwerte für die Entscheidung relevant sind, ist es sinnvoll, die Karten (des Computers) der Größe nach zu ordnen. Wir kennen dafür die Array-Methode sort(), die allerdings in-place sortiert, d.h. die Reihenfolge innerhalb des Arrays ändert.
Wir brauchen darum eine Kopie von *computerBlatt*, weil die Reihenfolge der Kartenwerte natürlich nicht geändert werden darf (sonst passen Kartenwerte und Karten-Objekte nicht mehr zusammen):

```
let blatt = computerBlatt; // FUNKTIONIERT NICHT!
blatt.sort(min2max); // FUNKTIONIERT AUCH NICHT!
```

Das sieht gut aus, ist aber falsch! Durch die Zuweisung wird kein neues Array mit den gleichen Werten erzeugt, sondern nur eine *Referenz*, d.h. ein weiterer Name für dasselbe Array. Durch das Sortieren von *blatt* werden die Werte in *computerBlatt* ebenfalls sortiert!
Zweiter Fehler: In Lektion 11 haben wir *sort()* mit dem Argument *min2max* eingesetzt. Das ist nur notwendig und führt auch nur dann zum richtigen Sortierergebnis, wenn das Array Zahlen enthält. Beim Sortieren von Zeichenketten (Kartenwerte!) muss *sort()* aber ohne Argument verwendet werden, damit richtig sortiert wird.

Beides ist im Quelltextes von ***computerEntscheidung***() korrekt eingebaut:

```
const computerEntscheidung = function () {
    // Karten sortieren
    let blatt = [];
    for (let wert of computerBlatt) {
        blatt.push(wert);
    }
    blatt.sort(); // von klein nach groß
    // Regel 1: Höchste Karte ausspielen
    if (start === "Computer") {
        return computerBlatt.indexOf(blatt[3]);
    }
    // Regel 2: Niedrigste Karte für Stichgewinn
    let gegenwert = spielerBlatt[spielerWahl];
    for (let wert of blatt) {
        if (wert > gegenwert) {
            return computerBlatt.indexOf(wert);
        }
    }
    // Regel 3: Niedrigste Karte ausspielen
    return computerBlatt.indexOf(blatt[0]);
}
```

Das ist die ganze „KI“. Sieht nach wenig aus, ist auch wenig, aber gehaltvoll. Es stecken wichtige Überlegungen darin und eine gute Implementierung. Und auch hier zeigt sich noch einmal, der Vorteil der Repräsentation der Karten in der Form "4R" usw.

Quax ist tatsächlich fertig! Zieht man Leerzeilen und Kommentarzeilen ab, dann hat der gesamte Quelltext einen Umfang von etwa 200 Zeilen – ein ***kurzes* Programm**. Dass es so kurz geworden ist, hat mindestens **vier Gründe**:

1. Es ist ein einfaches Spiel. Der Spielablauf und die Spiellogik sind nicht komplex. Für die „KI" des Computers benötigt man nicht mehr als 15 Zeilen.

2. Insgesamt wurde gut implementiert. Durch den geschickten Einsatz von eigenen Funktionen (für Teile, die mehrfach benötigt werden) und Schleifen, sowie geeignete Datenstrukturen ist der Code schlank geblieben.

3. Das Layout ist relativ schlicht. Abgesehen von den Karten, die optisch ganz passabel sind, wurde kein Wert auf ein ansprechendes Erscheinungsbild gelegt. Gerade solche kosmetischen Dinge blähen den Code aber auf.

4. Das Programm kommt (fast) vollständig ohne visuelle Effekte und Animationen aus. Auch das gehört im weiteren Sinne zur „Kosmetik" und lässt den Umfang des Programms schnell anwachsen.

Nachdem Quax nun fehlerfrei funktioniert, wollen wir uns überlegen, welche **Verbesserungsmöglichkeiten** es gibt. Das ist im übrigen generell ein **sinnvolles Vorgehen**: Sich *zunächst* auf das Wesentliche konzentrieren, bis ein Spiel fehlerfrei funktioniert. *Danach* dann aus einem spielbaren Spiel ein Kunstwerk machen.

Überlegungen zu den Spielregeln:
Quax habe ich mir ausgedacht, die Spielregeln auch. Man könnte sie also ändern und ein modifiziertes Quax daraus machen. Durch eine Zusatzregel könnte man z.B. erreichen, dass es immer einen Gewinner gibt: Wenn die Punktzahl am Ende gleich ist, dann gewinnt derjenige, der die höchste Karte in seinem Ablagestapel oder die meisten Stiche gewonnen hat.

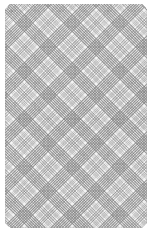
Man könnte den Gewinner auch generell nicht nach Punktsumme ermitteln, sondern nach der Anzahl der gewonnenen Stiche. Dann gibt es – bei fünf Stichen – immer einen Gewinner. Eventuell ist das gerechter?

Auf jeden Fall sollte (z.B. unterhalb des Spielfeldes) ein Link zu einer vollständigen, ordentlich als HTML-Dokument formatierten Spielanleitung stehen. Wer ein Spiel spielen will, muss die Spielregeln kennen!

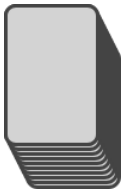
Überlegungen zum Layout:
Das **Spielfeld** sollte besser aussehen! Man sollte es als HTML-Canvas erstellen, zumindest einfärben, evtl auch eine geeignete Hintergrundgrafik verwenden, umrahmt mit abgerundeten Ecken und evtl. etwas Schatten.

Schön wäre es auch, wenn der **Name des Spiels** irgendwo stünde. In der Mitte des Spielfeldes ist am Anfang noch Platz – dort könnte man mit etwas Animation „Quax" ein-/ausblenden.

Die **Textelemente** könnten auf jeden Fall ansprechender gestaltet oder direkt durch Abbildungen ersetzt werden. Statt „Computer" und „Spieler" zu schreiben, könnte man passende Abbildungen an diesen Stellen einfügen. Der Rundzähler und der Informationstext zur Spielauswertung könnten ebenfalls eleganter gestaltet werden.



Die Vorderseiten der **Karten** sind passend für das Spiel gestaltet. Aber statt der grauen Rückseiten könnte man eine Grafik einsetzen, die stärker an die Rückseite einer echten Spielkarte erinnert.



Der **Ziehstapel** könnte als Stapel aus einzelnen Karten erkennbar sein. Beim Ziehen von Karten könnte die Höhe des Stapels dann abnehmen. Das lässt sich realisieren, indem man die einzelnen Abdrücke der Karten durchnummeriert und dann rückwärts nach und nach mit *`unstamp()`* entfernt.

Überlegungen zur Interaktion:
Das **Ausspielen der Karten** wäre realistischer, wenn der Spieler die Karte mit der Maus in die Mitte schieben könnte („drag & drop"). Das hätte – abgesehen von der aufwändigeren Implementation – allerdings den Nachteil, dass Quax auf Touch-Displays nicht spielbar wäre.

Das Programm kommt bisher fast ohne **Animation** aus. Lediglich die Bewegung der Karten beim Ziehen, Ausspielen und Ablegen ist sichtbar. Da geht noch was!
Man könnte etwa den „Denk- und Auswahlprozess" des Computers animieren (hier ist Kreativität gefragt), die Auswertung eines Stichs ebenso und die Präsentation des Endergebnisses auch.

Und für das **Aufdecken der Karten** bietet sich ein bisher noch nicht behandeltes Paar von Spider-Methoden an, die den Aufdeckprozess ziemlich gut animiert: ***`flipIn()`*** und ***`flipOut()`***. Sie bewirken das Umklappen des Spider-Objekts (= Karte) an einer frei wählbaren Kante von sichtbar zu unsichtbar bzw. von unsichtbar zu sichtbar. Wenn man in den Funktionen *zieheKarte()* und *computerZug()* die beiden fett gedruckten Zeilen ergänzt, dann wird das Aufdecken der Karten animiert:

```
karte.flipIn("middle"); // neu
karte.label = wert[0]; // alt
karte.color = farbe.get(wert[1]); // alt
karte.flipOut("middle"); // neu
```

Aufgaben

1. Setze einige der Verbesserungsvorschläge um und/oder verbessere die Spiellogik des Computers.