Serlo

Entscheidungen

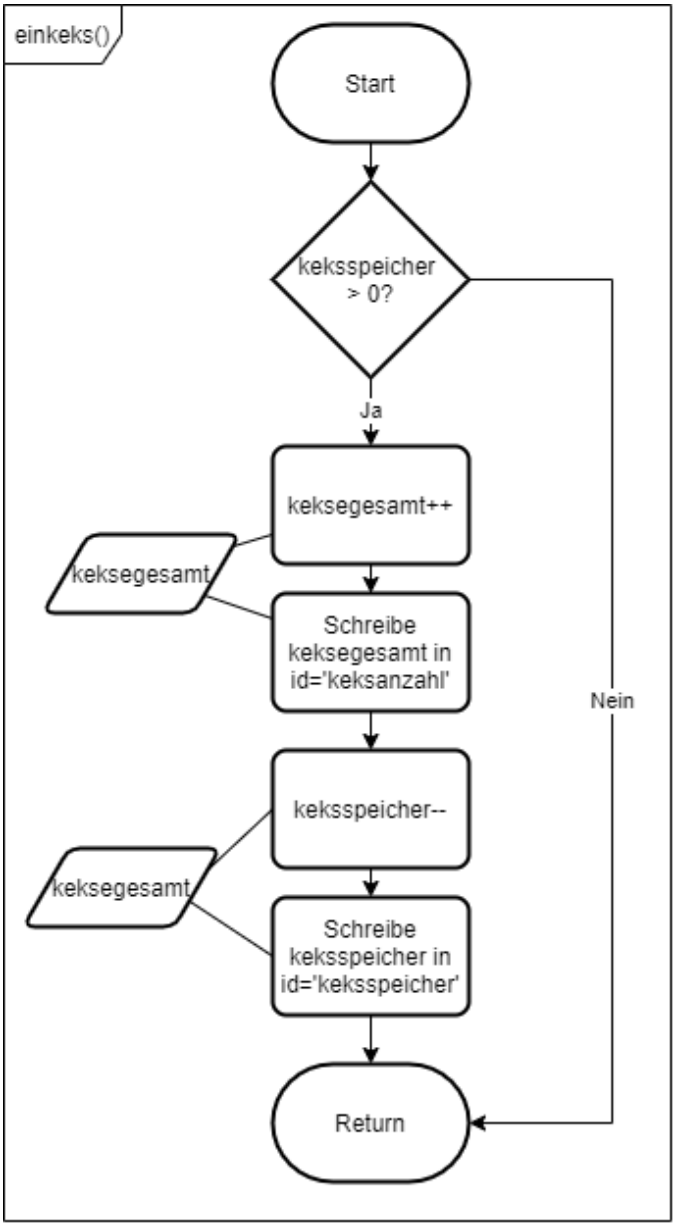
Einseitige Verzweigung

Bei einer Verzweigung handelt es sich um eine Kontrollstruktur. Bisher wurden alle Anweisungen in einem Code-Block von oben nach unten abgearbeitet. Mit Hilfe einer einseitigen Verzweigung kannst du festlegen, dass ein bestimmter Anweisungsblock (in geschweiften Klammern {}) nur ausgeführt wird, wenn eine **Bedingung** (in Runden Klammern ()) erfüllt ist. Du zeigst dem JavaScript Interpreter, dass eine Entscheidung getroffen werden muss, indem du das Signalwort **if** verwendest.

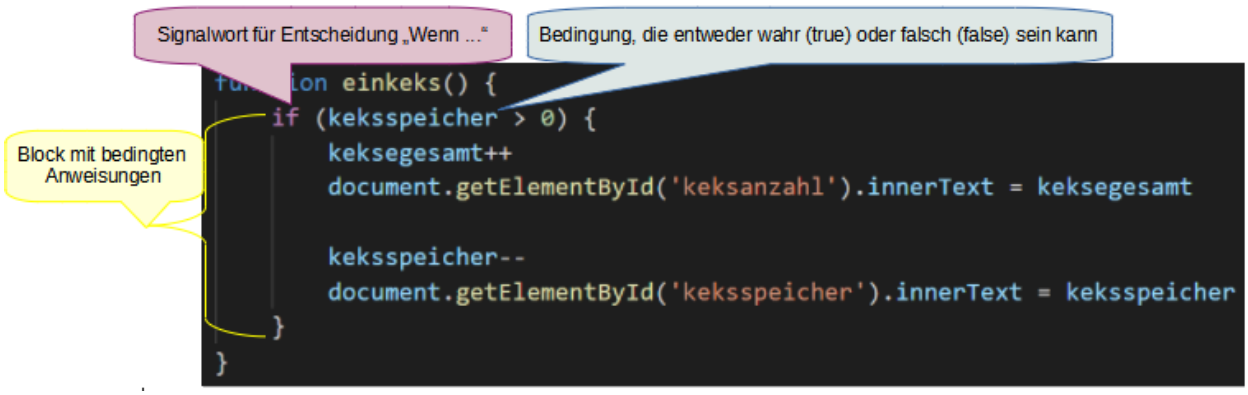
```
if (Bedingung){bedingter Anweisungsblock}
```

Beispiel Funktion einkeks(): Wenn die Bedingung *"keksspeicher > 0"* wahr ist (Ja-Zweig), dann werden die Anweisungen ausgeführt.

Bei einer einseitigen Verzweigung passiert im Nein-Zweig, also wenn die Bedingung nicht erfüllt wird, nichts.



Im JavaScript Code kannst du das folgendermaßen umsetzen



Wichtig ist dabei das Signalwort if (englisch für "wenn"). Dieses Signalwort findet sich in sehr vielen Programmiersprachen für genau diesen Zweck wieder.

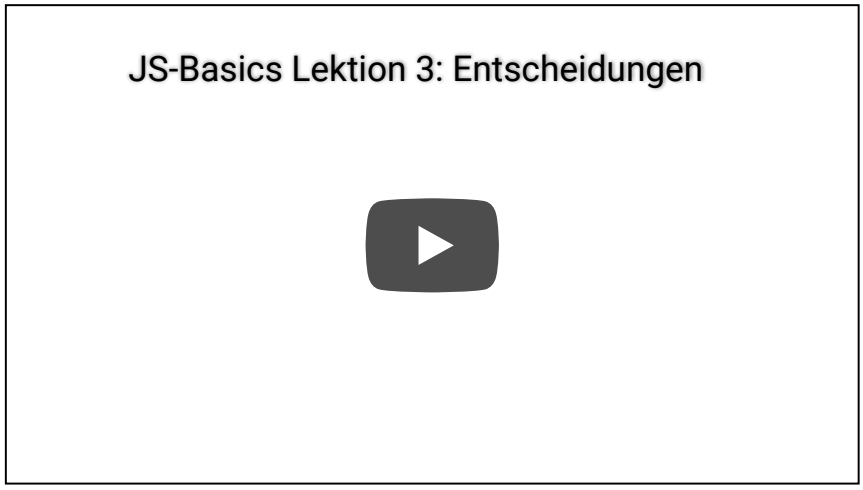
Hinweis zu Bedingungen

Für eine Bedingung muss der Interpreter nur in der Lage sein zu bestimmen, ob sie wahr oder falsch ist. Du kannst also z.B. auch Variablen mit dem Datentyp Boolean angeben. Eine Boolean Variable kann schließlich nur die Werte *true* oder *false* speichern.

Außerdem gibt es bestimmte andere Werte, die vom Interpreter generell als *true* oder *false* interpretiert werden. Da diese Werte dann nicht wirklich true oder false sind spricht man von *truthy* bzw. *falsy*. Das kann man sich oft sehr gut zu Nutze machen. Mehr Infos dazu bekommst du hier https://developer.mozilla.org/de/docs/Glossary/Truthy (https://developer.mozilla.org/de/docs/Glossary/Truthy) und hier https://developer.mozilla.org/de/docs/Glossary/Falsy (https://developer.mozilla.org/de/docs/Glossary/Falsy)

Aufgabe

Schau dir das Video bis zu Minute 5:41an und versuche das im Projekt direkt nach zu programmieren.



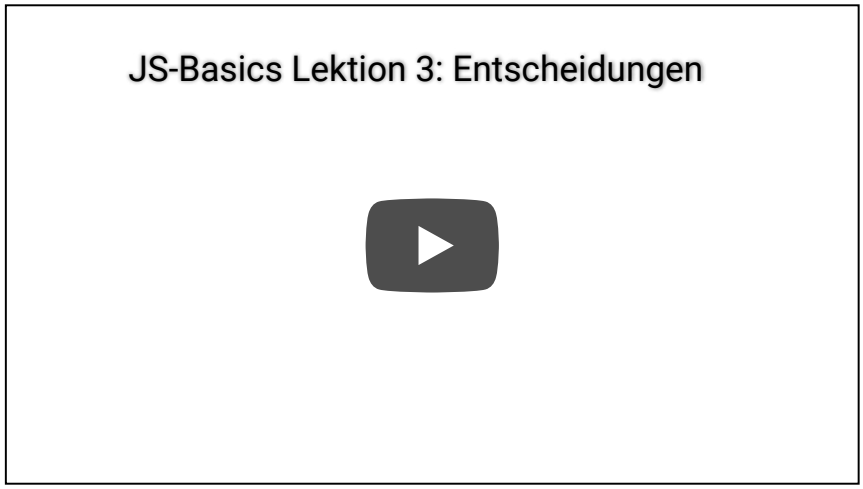
Vergleichsoperatoren

Um gute Bedingungen formulieren zu können, musst du oft Werte miteinander Vergleichen. Das geschieht mit Vergleichsoperatoren. Viele davon kennst du bestimmt schon aus der Mathematik. Damit lassen sich Werte, Variablen, Rückgabewerte von Funktionen etc. vergleichen. Ein Vergleich gibt immer einen Wahrheitswert zurück d.h. *true* oder *false*. Damit kann dann die Entscheidung getroffen werden, wie der Code weiter ausgeführt werden soll.

Die wichtigsten Vergleichsoperatoren in JavaScript findest du auf dieser Seite sehr gut zusammengefasst:
https://wiki.selfhtml.org/wiki/JavaScript/Operatoren/Vergleichsoperatoren (https://wiki.selfhtml.org/wiki/JavaScript/Operatoren/Vergleichsoperatoren)

Aufgabe

Lies dir die Quelle zum Thema Vergleichsoperatoren durch. Sieh dir anschließend das Video bis Minute 7:56 an. Ergänze den deinen Code im Projekt mit dem, was im Video gezeigt wird. Pausiere das Video und löse Aufgabe 1 selbständig im Projekt.



Zweiseitige Verzweigungen

Bei einer Zweiseitigen Verzweigung kommt zu unserem Anweisungsblock, der ausgeführt wird, wenn die Bedingung wahr ist, noch ein alternativer Anweisungsblock. Der wird ausgeführt, wenn die Bedingung falsch ist.

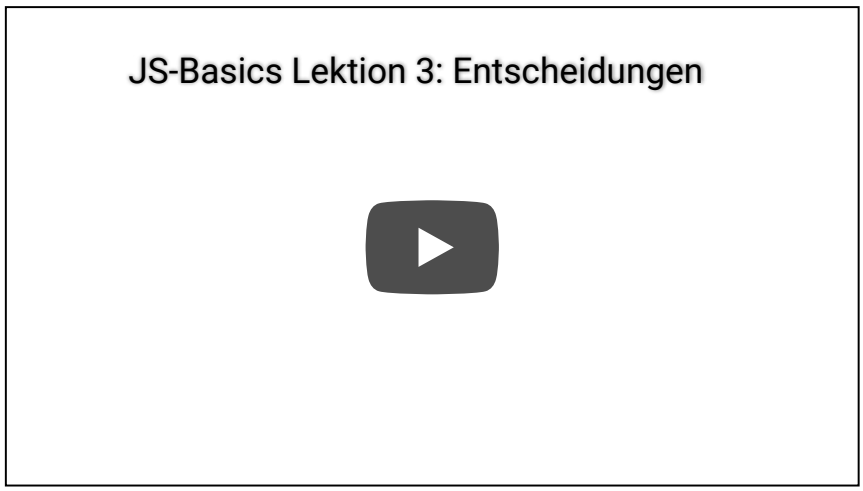
In natürlicher Sprache kann man solche Entscheidungen formulieren mit: Wenn *Bedingung*, dann *bedingte Anweisungen*, ansonsten *alternative Anweisungen.*

In JavaScript lässt sich das dann so ausdrücken:

```
if (Bedingung){
 bedingt Anweisung
} else {
alternative Anweisung
}
```

Aufgabe

Schau dir das Video bis Minute 10:30 an und programmiere das gezeigt in deinem Projekt nach.



Verschachtelungen

Code-Blöcke in JavaScript (gekennzeichnet durch geschweifte Klammern {}) können ineinander verschachtelt werden, d.h. man kann in einen Code-Block beliebige andere Code-Blöcke schreiben.

Bei Verzweigungen wird das besonders häufig verwendet, um komplexere Entscheidungen zu treffen. Man kann sich das vorstellen wie beim Autofahren, wo ich mehrmals hintereinander entscheide, ob ich links oder rechts abbiegen muss oder vielleicht einfach weiter gerade aus fahre.

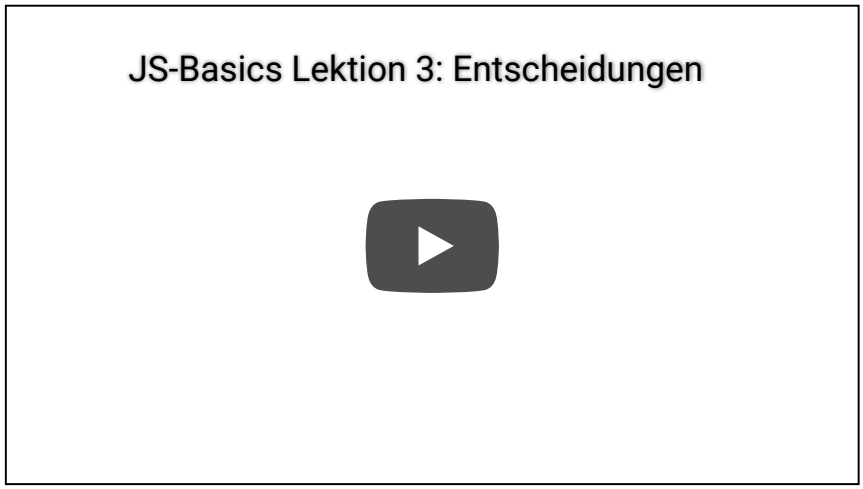
In JavaScript kann das dann so aussehen:

```
if (Bedingung1){
  bedingter Anweisungsblock1
  if(Bedingung2){
    bedingter Anweisungsblock2
  }
} else {
  alternativer Anweisungsblock1
  if(Bedingung3){
    bedingter Anweisungsblock3
  }
}
```

z.B. Anweisungsblock3 wird dann nur ausgeführt, wenn Bedingung1 falsch ist und Bedingung3 wahr.

Aufgabe

Schau dir das Video bis Minute 16:09 an. Programmiere das gezeigt nach. Löse anschließend Aufgabe 2 und überlege dir dann wo du noch sinnvolle Entscheidungen im Projekt einbauen kannst.



Anwendung: Schere-Stein-Papier

Dann ist es jetzt an der Zeit ein erstes Spiel umzusetzen, damit wir unseren keksspeicher auffüllen können. Wer die Regeln und Hintergründe dieses Spiels nicht kennen sollte findet z.B. bei Wikipedia hilfreiche Informationen: https://de.wikipedia.org/wiki/Schere,_Stein,_Papier (https://de.wikipedia.org/wiki/Schere,_Stein,_Papier)

Die Grafiken zum Spiel findest du im img Ordner des Projekts auf GitHub: https://github.com/baRockA/JS-Basic/tree/main/03_Entscheidungen/img (https://github.com/baRockA/JS-Basic/tree/main/03_Entscheidungen/img).

Aufgabe

Lade die Grafik-Dateien herunter und lege sie im img-Ordner deines Projektes ab.

HTML-Elemente hinzufügen

Damit das Spiel auf der Seite dargestellt wird müssen wir den HTML-Code der Seite erweitern.

- Zunächst benötigen wir ein neues Steuerelement im Element mit der *id="steuerung"*.
- Hier kannst du einfach ein Bild-Element anlegen, dass die Grafik *sspes.png* auf der Seite darstellt.
- Das Bild bekommt ein Event-Attribut *onclick*, dass die Funktion *oeffneSSP()* aufruft.

```
<img id="ssp" src="img/sspes.png" alt="Stein-Schere-Papier-Echse-Spock
```

Außerdem muss natürlich ein Bereich erstellt werden, in dem das Spiel angezeigt wird. Dieser sollte sich unter der Steuerung befinden.

- In diesem Bereich befinden sich zunächst auch Steuerelemente (Bilder) für Schere, Stein und Papier.
- Jedes Bild enthält ein Event-Attribut *onclick*, dass die Funktion *spieleSSP()* aufruft und für Schere die Zahl 1, bei Papier die Zahl 2 und bei Stein die Zahl 3

übergibt.

- Es folgt ein Bereich für den Showdown in dem das Ergebnis des Spiels, d.h. die Auswahl des Spielers und die des Gegners jeweils als Grafik dargestellt wird.
- Außerdem benötigen wir einen Absatz, um darin das Ergebnis zu verkünden.
- Zum Schluss des Dialogs gibt es noch einen Button, der *onclick* die FUnktion *schliesseSSP()* aufruft.
- Die Elemente, die später zur Ausgabe genutzt oder speziell mit CSS formatiert werden müssen erhalten eine eindeutige id.

```
<div id="ssp-dialog">
    <p>Wähle eine Option aus, um Kekse zu gewinnen.</p>
    <div><img src="img/1.png" alt="Schere" onclick="spieleSSP(1)">
    <div id="ssp-showdown">
        <div id="ssp-spieler">
            <h2>Deine Wahl</h2>
            <img id="ssp-spieler-wahl" src="img/.png" alt="Wahl de
        </div>
        <div id="ssp-gegner">
            <h2>Wahl des Gegners</h2>
            <img id="ssp-gegner-wahl" src="img/.png" alt="Wahl des
        </div>
    </div>
    <p id="ssp-ergebnis"></p>
    <button onclick="schliesseSSP()">Schließen</button>
</div>
```

Aufgabe

Übernimm den HTML-Code in dein Projekt und versuche alles zu verstehen. Teste das Projekt in einem Browser.

CSS Styling vornehmen

In der *default.css* Datei sorgen wir für eine schöne Darstellung unseres Spiels.

Auch hier stylen wir zunächst unser Steuerelement mit der *id="ssp"* in der Steuerung. Dabei orientieren wir uns an der Darstellung des Kekses.

```
#ssp {
    opacity: 0.5;
    width: 25%;
    grid-row: 1;
    grid-column: 1;
    justify-self: center;
    background-color: white;
    border-radius: 50%;
}

#ssp:hover {
    opacity: 1;
}
```

Dann brauchen wir noch etwas Formatierung für die Anzeige des Elements *id="ssp-dialog"* und des darin enthaltenen Elements *id="ssp-showdown"*

```
#ssp-dialog {
    display: none;
    position: absolute;
    background-color: rgba(255, 255, 255, 0.5);
    border: 5px solid white;
    margin: 5px;
    padding: 5px;
    text-align: center;
}

#ssp-showdown {
    background-color: rgba(255, 255, 255, 0.5);
    border: 3px solid orange;
    margin: 5px;
    padding: 5px;
    display: grid;
    grid-template-columns: 50% 50%;
}
```

Natürlich kannst du das Aussehen des Spiels hier beliebig verändern. Die wichtigste Eigenschaft für uns ist *display: none;* in *#ssp-dialog*. Darüber wird später unser Spiel ein bzw. ausgeblendet. Diese sollte also nicht verändert werden.

Aufgabe

Übernimm den CSS-Code in die Datei default.css und teste anschließend dein Projekt in einem Browser. Du kannst gerne mit den Eigenschaften arbeiten und die Darstellung nach deinen Wünschen ändern.

Externes Script einbinden

Um besonders umfangreiche Projekte übersichtlicher zu gestalten ist es sinnvoll JavaScript-Code in externe Dateien auszulagern. So ähnlich wie bei CSS lässt sich diese Datei dann über eine HTML-Element in die Seite einbinden.

Tatsächlich kannst du einfach ein <*script*>-Element ohne Inhalt nehmen und mit einem *src*-Attribut auf die Datei verweisen.

Eine JavaScript-Datei hat meistens die Dateiendung *.js*

```
<script src="js/ssp.js"></script>
```

Aufgabe

Erstelle im Projekt einen neuen Ordner mit dem Namen *"js"* und darin eine neue Datei mit dem Namen *"ssp.js".* Binde die Datei als Script am Ende deiner HTML-Datei ein wie im Beispiel gezeigt.

Schere-Stein-Papier mit If-Else

Zunächst brauchen wir in der *ssp.js* zwei einfache Funktionen, die den Spielbereich öffnen bzw. schließen. Die Funktionen *oeffneSSP()* und *schliesseSSP()* werden von den Steuerelementen auf der Seite bei einem Mausklick ausgeführt und setzen die display-Eiegenschaft des *ssp-dialog* beim Öffnen auf *'block'*, beim Schließen auf *'none'.*

Code

```
function schliesseSSP() {
    document.getElementById('ssp-dialog').style.display = 'none'
}

function oeffneSSP() {
    document.getElementById('ssp-dialog').style.display = 'block'
}
```

Die Funktion *spieleSSP()* bekommt als Parameter die Auswahl des Spielers (1: Schere, 2: Papier, 3: Stein) übergeben. Sie realisiert das eigentliche Spiel.

```
function spieleSSP(spielerWahl){ ... }
```

Mathematische Funktionen in JavaScript

Als erstes generieren wir mit der Funktion *Math.random()* eine Zufallszahl zwischen 0.0 und 1.0. Wir wollen aber eine Zahl zwischen 1 und 3. Darum multiplizieren wir den Rückgabewert der Funktion mit 3 und speichern das Ergebnis in einer lokalen Variable.

```
let computerWahl = Math.random() * 3
```

Jetzt haben wir aber leider Kommazahlen zwischen 0 und 3. Dieses Problem können wir lösen, indem wir zur computerWahl 0.5 addieren (dann haben wir Zahlen zwischen 0.5 und 3.5) und anschließend mit Hilfe der Funktion *Math.round()* die Zahl zu einer ganzen Zahl runden.

```
computerWahl = Math.round(computerWahl + 0.5)
```

Mehr Informationen dazu

Weitere Infos zu den mathematischen Funktionen gibt es hier: https://wiki.selfhtml.org/wiki/JavaScript/Objekte/Math

Ergebnis und Ausgabe

Damit wir uns für ein Ergebnis entscheiden können und das später ausgeben können, brauchen wir eine weitere lokale Variable *ergebnis*. Unser "Standardfall" wäre ein Unentschieden, deshalb speichern wir den Text, den wir bei einem Unentschieden ausgeben wollen in der lokalen Variable.

Code

```
let ergebnis = "Unentschieden!"
```

Spiel-Entscheidungen

Als erstes prüfen wir, ob ein Unentschieden vorliegt. Das würde bedeuten, dass *computerWahl* und *spielerWahl* den selben Wert hätten. Wenn das der Fall wäre, dann wird der *keksspeicher* inkrementiert. Ansonsten müssen wir uns entscheiden, ob der Computer oder der Spieler gewonnen hat.

Code

```
if (computerWahl == spielerWahl) {
    keksspeicher++;
} else {
    ...
}
```

Um zu entscheiden wer gewonnen hat genügt es zu entscheiden, ob der Computer gewonnen hat. Ansonsten hat natürlich der Spieler gewonnen. In beiden Fällen muss der Text in der Variable *ergebnis* durch Zuweisung eines neuen Werts angepasst werden. Wenn der Spieler gewonnen hat, wird der *keksspeicher* um 30 erhöht.

Die Bedingung wird nun etwas komplexer, denn Sie besteht aus mehreren Vergleichen der Werte von *computerWahl* und von *spielerWahl*, die logisch miteinander verknüpft werden.

Als Text würde die Bedinung so aussehen: Wenn computerWahl ist 1 UND spielerWahl ist 2 ODER computerWahl ist 2 UND spielerWahl ist 3 ODER computerWahl ist 3 UND spielerWahl ist 1, dann hat der Spieler verloren.

Logische Verknüpfung in JavaScript wird durch die Ziechen && für UND und || für ODER vorgenommen. Es gibt natürlich noch weitere logische Verknüpfungen. Siehe dazu: https://wiki.selfhtml.org/wiki/JavaScript/Operatoren/Logische_Operatoren (https://wiki.selfhtml.org/wiki/JavaScript/Operatoren/Logische_Operatoren)

Code

```
        //Wenn der Computer gewinnt ...
        if (computerWahl == 1 && spielerWahl == 2 ||
            computerWahl == 2 && spielerWahl == 3 ||
            computerWahl == 3 && spielerWahl == 1) {
            ergebnis = "Leider verloren... keine Kekse für dich"
        } else {
            // ... ansonsten gewinnt der Spieler
            ergebnis = "Super, gewonnen! 30 Kekse für dich!"
            keksspeicher += 30
        }
```

Ausgabe auf der Seite

Dann müssen wir das Ergebnis nur noch auf der Seite darstellen. Da die Grafiken richtig nummeriert sind können wir die Werte der Variablen *computerWahl* und *spielerWahl* für die Auswahl der richtigen Grafik im src-Attribut der Elemente mit der id *ssp-gegner-wahl* bzw. *ssp-spieler-wahl* nutzen. Achte dabei aber auf die Konkatenation mit der richtigen Pfadangabe.

Den Wert der Variable *ergebnis* geben wir als Text im Element mit der id *ssp-ergebnis* aus.

Code

```
document.getElementById('ssp-gegner-wahl').src = 'img/' + computerWah
document.getElementById('ssp-spieler-wahl').src = 'img/' + spielerWah
document.getElementById('ssp-ergebnis').innerText = ergebnis
```

+ Kursseite hinzufügen