JavaScript

Agenda

JavaScript Einführung

Variablen

Datentypen

Syntax & Konstrukte

OOP in JavaScript

Module

DOM

Sync & Async

JavaScript Einführung

Was ist JavaScript?

- Programmiersprache für Webentwicklung
  - HTML: Strukturierung von Inhalt
  - JavaScript: Steuerung des Verhaltens
- ECMAScript (ES<number>): Spezifizierung von Standards für eine Sprache
  - Dient als Basis für die JavaScript Implementierung
  - Verschiedene Versionen von JavaScript basieren auf verschiedenen ECMAScript Versionen
- Asynchron (non-blocking Code)
- JavaScript Runtime Support wurde Stück für Stück in die gängigen Browser integriert

JavaScript Geschichte

1995

Erfindung von JavaScript durch Brendan Eich

Für den Browser Netscape.

1997

Aus JavaScript entstand der ECMA Standard (Version ES1)

JavaScript wurde fortan immer auf Basis des aktuellen ECMAScript Standards weiterentwickelt.

2005

Entstehung der Technologie Ajax und damit Boom

Mit Ajax ließen sich erstmals Inhalte im Hintergrund laden (dynamische Websites).

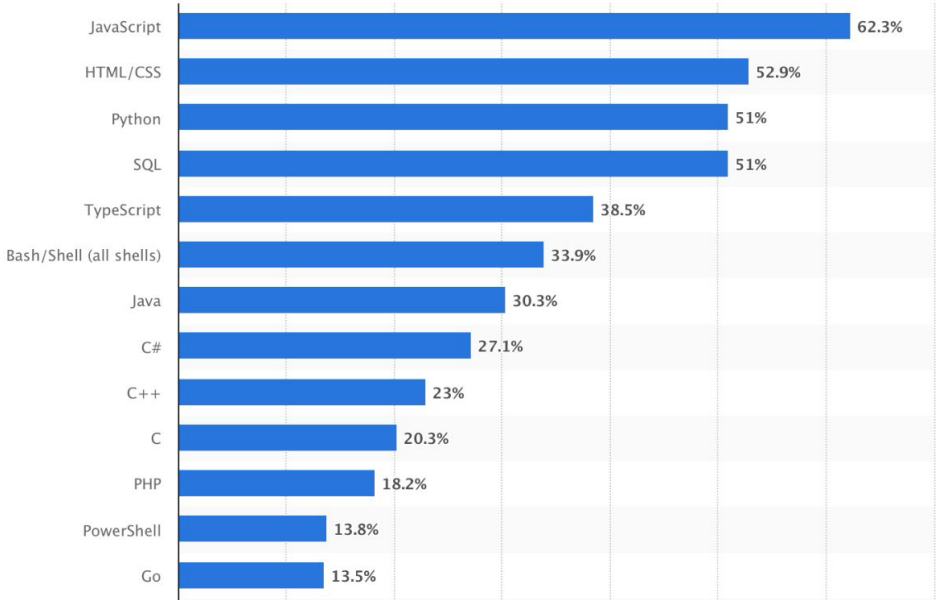
2009

Erfindung von NodeJS

Von da an konnte JavaScript nicht nur auf dem Client, sondern auch auf dem Server ausgeführt werden.

Beliebtheit

Most used programming languages among developers worldwide as of 2024



DOWNLOAD

PDF + XLS + PNG + PPT +

Source
- Show sources information
- Show publisher information
- Use Ask Statista Research Service

Release date
July 2024

Region
Worldwide

Survey time period
May 19 to June 20, 2024

Number of respondents
60,171 respondents

JavaScript ausführen

- NodeJS (Backend Implementierung)
  - Software NodeJS (Laufzeitumgebung) muss auf Server installiert werden
  - Typischerweise werden Anfragen vom Server verarbeitet, bei denen z.B. eine Anfrage auf einen bestimmten Port des Servers mit bestimmten Parametern eingeht
- Einbettung in HTML

```
<html>
  <head>
    <title>JavaScript Examples</title>
  </head>
  <body>
    <script>console.log('Hello World!');</script> <!-- direct execution of commands in HTML -->
    <script src="hello_world.js"></script> <!-- execution of commands in external file -->
  </body>
</html>
```

Variablen

Variablen + Konstanten

- Namenskonvention: camelCase

```
let firstName  = "Heide";  // variable
const lastName = "Witzka"; // constant
```

- Deklaration von Variablen:
  - Old (< ES6): **var** als Keyword für Deklaration von Variablen
  - New (>= ES6): **let** als Keyword für Deklaration von Variablen (var wird noch unterstützt)

```
if (true) {
  var functionScopeVariable = "test"; // variable with function scope
  let blockScopeVariable = "test"; // variable with block scope
}
console.log(functionScopeVariable); // test
console.log(blockScopeVariable); // ReferenceError: blockScopeVariable is not defined
```

- Name darf kein Reserved Keyword sein (https://www.w3schools.com/js/js_reserved.asp)

Datentypen

Primitive Types

- String
  - Template Literals
  - Concatenation
- Number
- Boolean
- undefined
- null

```
let firstName = "Hella";    // String
let lastName  = 'Wahnsinn'; // String
let fullName  = firstName + " " + lastName; // concatenated String
let fullNameLiteral = `${firstName} ${lastName}`; // Template literal
let age       = 30;         // Number
let isAdult   = true;       // Boolean
let child     = undefined;  // undefined
let spouse    = null;       // null
```

Reference Types

```
let person = {
    firstName: "Wilma",
    lastName: "Ruhe",
    age: 45,
    getFullName: function() {
        return this.firstName + " " + this.lastName;
    }
};
console.log(person.lastName); // accessing object via dot notation
console.log(person["lastName"]); // accessing object via bracket notation
person.lastName = "Bier"; // changing object property
person.getFullName(); // calling object method
```

Objekt

- Besteht aus “Key/Value” Paaren
- Kann Eigenschaften und Methoden haben
- Zugriff auf Eigenschaften via Punkt-/ oder Klammernotation
- “this” in Objektmethoden bezieht sich auf das Objekt, von dem aus die Methode aufgerufen wird (Funktionskontext)

Reference Types

```
function greetPerson(firstName, lastName) {
    console.log("Hello " + firstName + " " + lastName);
}
function calculateSum(a, b) {
  return a + b;
};

geetPersonArrow = (firstName, lastName) => {
  console.log("Hello " + firstName + " " + lastName);
}
greetPerson("Otto", "Päde"); // calling function --> Hello Otto Päde
let sumResult = calculateSum(1, 2); // calling function --> 3
console.log(sumResult); // 3
```

Funktion

- Wiederverwendbare Code-Blöcke
- Mit verschiedenen Parametern aufrufbar
- Wird mit () ausgeführt
- Arrow-Funktionen erlauben kürzere Syntax
    - Vorsicht: “this” bedeutet etwas anderes in “normalen” Funktionen (anderer Kontext)
- Mit “return” werden Wert(e) von der Funktion zurückgegeben und sie damit beendet

Reference Types

```
// Array
let firstNames = ["Wilma", "Heide", "Otto", "Rosi", "Kai"];
console.log(firstNames[0]); // accessing first array element via index
firstNames[0] = "Willma"; // changing array element
console.log(firstNames.length); // length of array

firstNames.push("Rainer"); // adding array element
firstNames.pop(); // removing last array element
firstNames.shift(); // removing first array element
firstNames.unshift("Wilma"); // adding array element at the beginning
firstNames.splice(1, 1); // removing array element at index 1
firstNames.splice(1, 0, "Heide"); // adding array element at index 1
```

Array

- Ist eine Liste mit bestimmten Werten (können Werte von verschiedenem Typ sein!)
- Wird üblicherweise genutzt, um programmatisch mehrere Werte zu vergleichen
- Ist ein spezielles Objekt in JavaScript → hat Attribute und Funktionen

Typing

Dynamic Typing

- Datentyp von Variablen ist beliebig änderbar
- Ermöglicht leichteres Lernen der Sprache
- Leichtere Implementierung von komplexen Problemen

```
let firstName = "Julian";
console.log(typeof firstName); // string
firstName = 30;
console.log(typeof firstName); // number
firstName = true;
console.log(typeof firstName); // boolean
```

Static Typing

- Datentyp von Variablen ist fest definiert und die Zuweisung eines Werts anderen Datentyps wird als Fehler erkannt (Typsicherheit)
- Programmierfehler können leichter erkannt werden

Exkurs: TypeScript

- Syntaktisches “Superset” für JavaScript
- TypeScript Compiler wandelt den geschriebenen Code in JavaScript um
- Gleiche Syntax wie JavaScript mit der zusätzlichen Möglichkeit feste Typen zu definieren
- Meldet Fehler, wenn z.B. einer Variable, welche als “String” definiert ist, ein Wert vom Typ “Number” zugewiesen wird

→ Typsicherheit (zur Laufzeit)

```
let firstName: string = "Rosi";
let lastName: string = "Ne";
lastName = 1; // Error: Type 'number' is not assignable to type 'string'.
```

Übung

Ein/e Student/in war gerade dabei JavaScript zu lernen. Er / sie wollte eine Funktion schreiben, die einen Namen (string) als Parameter akzeptiert und eine zufällige Zahl generiert und , je nachdem ob diese Zahl gerade oder ungerade ist, mit dem String " ist ein Depp." oder mit dem String "ist ein Genie." verkettet und zurückgibt. Diese Funktion soll dann mehrfach hintereinander mit verschiedenen Namen als Argument ausgeführt werden. Könnt ihr ihm dabei helfen?

Tipp:

- Zufallszahl zwischen 0 + 10 mit: `Math.floor(Math.random() * 10);`
- Modulo Operator für die Ermittlung von gerade / ungerade nutzen

Syntax & Konstrukte

Kommentare

- werden vom Interpreter ignoriert
- über mehrere Zeilen möglich
- werden oft zur Dokumentation für die Verständlichkeit des Codes verwendet

```
// Single Line Comment

/* Multi
    Line
    Comment */
```

Bedingungen

```
let age = 18;
if (age >= 18) {
    console.log("You are an adult!");
} else if (age >= 14) {
    console.log("You are a teenager!");
} else {
    console.log("You are a child!");
}
```

```
let day = 3;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Unknown day");
}
```

- Ausführen von verschiedenen Aktionen anhand verschiedener Bedingungen
- **If / Else:**
  - Einfaches “wenn/dann” Konstrukt
  - Mehrere Bedingungen aufeinanderfolgend möglich
  - Verkettung von Bedingungen mit “and” und “or”
- **Switch:**
  - Ausführen von verschiedenen Aktionen anhand verschiedener Bedingungen
  - Expression im switch Statement wird einmalig ausgeführt
  - Wenn ein Case mit der ausgeführten Expression übereinstimmt, wird der jeweilige Code Block ausgeführt
  - Wenn kein Case übereinstimmt: “default” Block wird ausgeführt

Operatoren

Rechenoperatoren

```
let a = 5;
let b = 10;

console.log(a + b); // 15
console.log(a - b); // -5
console.log(a * b); // 50
console.log(a / b); // 0.5
console.log(a % b); // 5
```

Vergleichsoperatoren

```
let num = 5;

console.log(num === 5); // equal value and equal type --> true

console.log(num == "5"); // equal value --> true

console.log(num !== 5); // not equal value or not equal type -->false

console.log(num != "5"); // not equal value --> false

console.log(num > 5); // greater than --> false

console.log(num >= 5); // greater than or equal --> true

console.log(num < 5); // less than --> false

console.log(num <= 5); // less than or equal --> true
```

Operatoren

Logische Operatoren

```
let age = 18;
let isAlcoholic = false;
if (age >= 18 && !isAlcoholic) {
    console.log("You are allowed to drink!");
}

if (!age >= 18 || isAlcoholic) {
    console.log("You are not allowed to drink!");
}
```

while-Schleifen

- while:
  - Führe bestimmte Befehle aus solange eine Bedingung erfüllt ist
  - Bedingung wird **vor** Ausführung des Codes geprüft
- do-while:
  - Führe bestimmte Befehle aus solange eine Bedingung erfüllt ist
  - Bedingung wird **nach** Ausführung des Codes geprüft

```
let i = 0;
while (i < 10) {
    console.log(i);
    i++;
}
```

```
let i = 0;
do {
    console.log(i);
    i++;
} while (i < 10);
```

for-Schleifen

- for-in:
  - Üblicherweise nur für Objekte (möglich für Arrays, aber fehleranfällig)
  - Überspringt leere Elemente
- for-of:
  - Zugriff auf den Wert des Array Elements
  - Kein Zugriff auf den Index des Array Elements
  - Ignoriert nicht-numerische Properties des Arrays
- for:
  - Generelle Schleife, um Befehle wiederholt auszuführen
  - Muss nicht auf ein Array bezogen sein
  - Ignoriert nicht-numerische Properties des Arrays
- forEach:
  - Ist eine Funktion des JavaScript Array Objekts (Das Array ist auch ein spezielles Objekt in JavaScript)
  - Ignoriert nicht-numerische Properties des Arrays
  - Überspringt leere Elemente
  - Hat einen anderen Funktionskontext (es sei denn man nutzt Arrow Function)
  - Nicht kompatibel mit async/await oder Generators

```
let person = {
    firstName: "Wilma",
    lastName: "Ruhe",
    age: 45
};
for (let property in person) {
    console.log(property); // firstName, lastName, age
    console.log(person[property]); // Wilma, Ruhe, 45
}
```

```
let firstNames = ["Wilma", "Heide", "Otto", "Rosi", "Kai"];
for (let firstName of firstNames) {
    console.log(firstName);
}
```

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```

```
let firstNames = ["Wilma", "Heide", "Otto", "Rosi", "Kai"];
firstNames.forEach(function(firstName) {
    console.log(firstName);
});
```

Break & Continue

- Break beendet eine Schleife manuell
- Continue springt zum nächsten Schleifendurchlauf

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break;
    }
    console.log(i);
}
```

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        continue;
    }
    console.log(i);
}
```

Error Handling

```
try {
  console.log("Hello World!");
} catch (error) {
  console.log(error);
  throw("Error!");
} finally {
  console.log("I get executed anyway!");
}
```

- In “try” Block wird auszuführender Code definiert
- In “catch” Block werden Fehler abgefangen und behandelt
- In “finally” Block wird Code definiert, der unabhängig vom Fehlerstatus ausgeführt wird
- Mit “throw” können eigens definierte Fehler ausgegeben werden (String, Boolean, Number / Objekt)
- JavaScript beinhaltet eigene Fehlerklassen, die genutzt werden können und die dem User hilfreiche Information für die Ursache des Fehlers geben

JSON

- Enthält “Name/Value” Paare (ähnlich wie JS Objekte)
- Name immer in doppelten Anführungszeichen
- Array wird in eckigen Klammern definiert ; kann Objekte beinhalten
- JS Objekte sind JSON kompatibel

```
let person = {
    firstName: "Wilma",
    lastName: "Ruhe",
    age: 45
};
let personJson = JSON.stringify(person);
console.log(personJson);
let personObject = JSON.parse(personJson);
console.log(personObject);
```

{} json-example.json json-example.json/...

```
{
    "persons": [
        {
            "firstName": "Marie",
            "lastName": "Juana",
            "age": 25
        },
        {
            "firstName": "Hella",
            "lastName": "Wahnsinn",
            "age": 30
        },
        {
            "firstName": "Jo",
            "lastName": "Ghurt",
            "age": 45
        }
    ]
}
```

Übung

Wir spielen gerade eine Runde Scrabble. Wir wollen nun eine Funktion schreiben, die uns die maximal mögliche Punktzahl berechnet, die wir mit unserer aktuellen Hand erreichen können (Summe berechnen). Eine Hand ist ein Array aus 7 Objekten mit den Eigenschaften “tile” und “score”:

```
[
  { tile: "N", score: 1 },
  { tile: "K", score: 5 },
  { tile: "Z", score: 10 },
  { tile: "X", score: 8 },
  { tile: "D", score: 2 },
  { tile: "A", score: 1 },
  { tile: "E", score: 1 }
]
```

OOP in JavaScript

Einleitung OOP

- Basiert auf Objekten (wie vorher in Reference Types erklärt)
- Klassen sind Baupläne für Objekte mit Eigenschaften und Methoden
- Jedes Objekt, das aus dieser Klasse erstellt wird, hat dann diese Eigenschaften und Methoden
- Vererbung: Erstellung von Klassen, die als Basis für weitere Klassen verwendet werden können

Objekte und ihr Kontext

- Objekte vom Typ “Object” sind beliebig veränderbar
- “this” in Methoden von Objekten bezieht sich immer auf das aktuelle Objekt

```
let person = {
    firstName: "Peter",
    lastName: "Silie",
    age: 45,
    getFullName: function() {
        return this.firstName + " " + this.lastName;
    }
}; // Object Literal of type Object
console.log(person.firstName); // Peter
console.log(person["firstName"]); // Peter
person.gender = "male";
console.log(person.gender); // male
console.log(person.getFullName()); // Peter Silie
```

Konstruktoren

- Konstruktoren sind normale Funktionen
- Mit “new” können neue Objekte mit diesem Konstruktor erstellt werden

```
// Constructor Function
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.getFullName = function() {
      return this.firstName + " " + this.lastName;
  };
}
let personA = new Person("Rainer", "Ernst");
console.log(typeof personA); // object
console.log(personA.getFullName()); // Rainer Ernst
```

Prototypen

- Konstruktorfunktionen bekommen immer die Eigenschaft “prototype”
- Darüber können nachträglich Eigenschaften und Methoden für alle Objekte, die mit diesem Konstruktor erstellt wurden, angelegt werden

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
Person.prototype.getFullName = function() {
    return this.firstName + " " + this.lastName;
};
let personA = new Person("Kai", "Sehr");
console.log(personA.getFullName()); // Kai Sehr
```

Klassen

- Wird im Hintergrund zu Konstruktoren und Prototypes übersetzt (syntaktischer Zucker)
- “class” leitet das Erstellen einer Klasse ein
- “constructor” ist das Keyword für die Konstruktorfunktion

```
class Person {
  constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
  }
  getFullName() {
      return this.firstName + " " + this.lastName;
  }
  setlastName(lastName) {
    this.lastName = lastName;
  }
}
let person = new Person("Rosi", "Ne");
console.log(person.getFullName()); // Rosi Ne
person.setlastName("Ja");
console.log(person.getFullName()); // Rosi Ja
```

Vererbung

- “extends” erweitert die Basisklasse
- “super” ermöglicht Zugriff aus Basisklasse
- Mit super im Konstruktor wird der Konstruktor der Basisklasse aufgerufen
- Super kann verwendet werden, um auf Eigenschaften oder Methoden der Basisklasse direkt zuzugreifen

```
class Finanzbeamter extends Person {
  constructor(firstName, lastName, salary) {
      super(firstName, lastName);
      this.salary = salary;
  }
  getSalary() {
      return this.salary;
  }
}
class Influencer extends Person {
  constructor(firstName, lastName, followers) {
      super(firstName, lastName);
      this.followers = followers;
  }
  getFollowers() {
      return this.followers;
  }
}
let finanzbeamterA = new Finanzbeamter("Rainer", "Ernst", 5000);
console.log(finanzbeamterA.getFullName()); // Wilma Ruhe
console.log(finanzbeamterA.getSalary()); // 5000
let influencerA = new Influencer("Lilli", "Putaner", 10);
console.log(influencerA.getFollowers()); // 10
```

Übung

Schreibe eine Klasse “Fahrzeug”, die die Eigenschaften “Hersteller”, “Modell” und “Baujahr” und eine Methode zum Anzeigen der Fahrzeugdetails beinhaltet. Schreibe dann eine weitere Klasse “Auto”, die von der Klasse “Fahrzeug” erbt und zusätzlich die Eigenschaft “AnzahlTüren” besitzt. Ebenso muss für diese Klasse die Methode zum Anzeigen der Fahrzeugdetails überschrieben werden. Erstelle dann jeweils ein Objekt der beiden Klassen und lasse die Fahrzeugdetails in der Konsole ausgeben.

Module

Exports & Imports

- Code kann auf beliebig viele Dateien verteilt werden
- Code muss aus Datei exportiert werden, damit er in anderer Datei importiert werden kann
- **Named Exports:** können mehrfach pro Datei implementiert werden
- **Default Export:** kann nur exakt einmal pro Datei existieren

JS default-export-example.js default-export-example.js/ testD

```
1   // default export
2   export default function testDefaultExport() {
3       console.log("Hello World!");
4   }
```

JS named-export-example.js named-export-example.js/...

```
1   // named export
2   export function testNamedExport() {
3       console.log("Hello World!");
4   }
5   function testNamedExport2() {
6       console.log("Hello World!");
7   }
8   let testNamedExport3 = 'Hello World!';
9   export { testNamedExport2, testNamedExport3 };
```

Imports

- **Named Exports:** müssen mit dem exakt gleichen Namen importiert werden mit Destrukturierung
- **Default Export:** kann nur exakt einmal pro Datei existieren; kann ohne Destrukturierung importiert werden mit beliebigen Namen

```
import myExportedFunction from './default-export-example.js';
import {
  testNamedExport,
  testNamedExport2,
  testNamedExport3
} from './named-export-example.js';
```

Übung

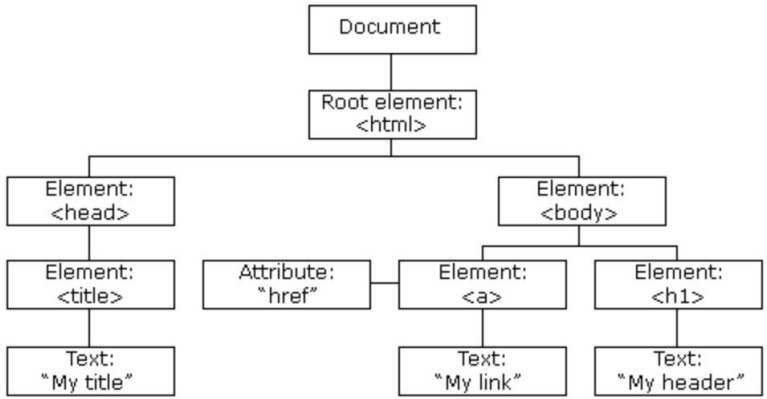
Lege zwei “.js” Dateien an und exportiere eine Variable und eine Funktion, die etwas in die Konsole schreibt, aus der einen und importiere Sie in die andere Datei und rufe sie dort auch auf. Die Datei, in der importiert wird, soll in eine einfache HTML Datei als script-Block integriert werden. Rufe die HTML Seite im Browser auf und prüfe in der Konsole, ob die Ausgabe erfolgt ist.

Tipp: HTML script tag: type=”module”

DOM (Document Object Model)

Was ist das DOM?

- Steht für Document Object Model
- Standard für den Zugriff auf Dokumente (z.B. für HTML)
- Beinhaltet Elemente als Objekte und deren Eigenschaften, Methoden und Events



DOM Methoden

- Methoden, die für HTML Elemente ausführbar sind
- Beispiele:
  - getElementById(elementId): Zugriff auf bestimmtes HTML Element mit einer ID
  - getElementsByClassName(className): gibt eine Liste von Elementen mit dieser Klasse zurück
  - innerHTML():Zugriff auf Inhalt eines HTML Elements

```
<body>
    <button id="myBtn" class="myButtons">Try it</button>
    <button id="myBtn2" class="myButtons">Try it 2</button>
</body>
<script>
    let buttonOne = document.getElementById("myBtn"); // get the element with id="myBtn"
    console.log(buttonOne.innerHTML); // get the innerHTML of the element with id="myBtn"
    let buttons = document.getElementsByClassName("myButtons"); // get all elements with class="myButtons"
    for (let i = 0; i < buttons.length; i++) {
        console.log(buttons[i].innerHTML); // get the innerHTML of the element with class="myButtons"
    }
</script>
```

DOM Formulare

- HTML Formulare können mit der DOM validiert werden

```
<script>
    function validateForm() {
        let x = document.forms["myForm"]["fname"].value;
        if (x == "") {
            alert("Name must be filled out");
            return false;
        }
    }
</script>
```

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
    Name: <input type="text" name="fname">
    <input type="submit" value="Submit">
</form>
```

DOM CSS

- Style von HTML Elementen kann durch DOM Manipulation verändert werden

```
<p id="p2">Hello World!</p>

<script>
    document.getElementById("p2").style.color = "blue";
</script>
```

DOM Events

```
<h1 onclick="changeText(this)">Click on this text!</h1>

<script>
    function changeText(id) {
        id.innerHTML = "Ooops!";
    }
</script>
```

- Mithilfe von JavaScript kann auf Events reagiert werden
- Beispiele für unterstützte Events:
  - onload, onunload: Benutzer betritt oder verlässt die Seite
  - oninput: tritt auf, wenn der User eine Eingabe macht
  - onchange: tritt auf, wenn sich ein Element verändert hat (nach der Eingabe)
  - onmouseover, onmouseout: tritt auf, wenn sich die Maus über ein Element oder vom Element weg bewegt
  - onmousedown, onmouseup: tritt auf, wenn die Maus geklickt, oder der Klick losgelassen wird
  - DOMContentLoaded: tritt auf, wenn die komplette DOM von JS geladen wurde (Document Event)
- Events können auch mit addEventListener() HTML Elementen zugewiesen werden

Übung

Schreibe eine HTML Website, die eine Zahl darstellt und einen Button beinhaltet, der bei einem Klick die Zahl inkrementiert. Die Änderung sollte unmittelbar sichtbar sein. Der Startwert der Zahl ist 0. Zusätzlich soll bei jedem Klick die Farbe der angezeigten Zahl verändert werden. Immer, wenn die Zahl durch 7 teilbar ist, soll ein Alert ausgegeben werden. Zusätzlich sollte ein Button existieren, der die Zahl dekrementiert und auch entsprechend die Farbe verändert und ein Button, der den Zähler und die Farbe zurücksetzt.

Tipp: Zufällige Farbe mit: `Math.floor(Math.random()*16777215).toString(16);`

Sync & Async

Callbacks

- Callback ist eine Funktion, die als Argument in eine andere Funktion übergeben wird und ausgeführt wird, wenn die erste Funktion ausgeführt wurde
- Kein eingebautes Error Handling
- Schlecht lesbar (Callback Hell)

```
function greet(callback) {
    setTimeout(function() {
        callback("Hello World!");
    }, 2000);
}

greet(function(result) {
    console.log(result);
});
```

Promises

- Promise ist ein spezielles Objekt, das die eventuelle Fertigstellung oder Abbruch einer asynchronen Operation darstellt und dem daraus resultierenden Ergebnis
- Ermöglicht vielfache Verkettung von asynchronen Operationen
- Einfacher zu lesen als Callbacks
- Einfache Fehlerbehandlung
- “then” wird ausgeführt, wenn das Promise erfüllt ist
- “catch” wird ausgeführt, wenn das Promise fehlschlägt

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve("Hello World!");
    }, 2000);
});

promise.then(function(result) {
    console.log(result);
}).catch((error) => {
  console.log(error);
});
```

Async / Await

- Ermöglicht es asynchronen Code zu schreiben, der wie synchroner Code aussieht (neuste Variante)
- “async” kennzeichnet eine Funktion, in der asynchrone Operationen ausgeführt werden
- “await” kennzeichnet, dass auf die Erfüllung eines Promises gewartet werden soll, bevor der folgende Code ausgeführt wird

```
async function testAsyncAwait() {
  let promise = new Promise(function(resolve, reject) {
      setTimeout(function() {
          resolve("Hello World!");
      }, 2000);
  });

  let result = await promise;
  console.log(result);
}
```

Fetch Api

- JavaScript Schnittstelle um HTTP Requests auszuführen und zu verarbeiten
- Über den Parameter method können die unterschiedlichen Methoden (*'POST'*, *'PUT'*, *'DELETE'*, etc) ausgewählt werden
- Mehr Infos gibt es in der Dokumentation

```
const url = "https://api.kanye.rest/";
try {
  const response = await fetch(url, { method: "GET" });
  if (!response.ok) {
    throw new Error(`Response status: ${response.status}`);
  }

  const json = await response.json();
  console.log(json);
} catch (error) {
  console.error(error.message);
}
```

Übung

Erstelle eine Website, die die öffentliche API https://pokeapi.co/docs/v2#pokemon-section konsumiert und eine Liste von Pokemon in Tabellenform darstellt. Dargestellt werden sollte der Name und z.B. die Moves eines Pokemon.

Tipps:

- fetch zur Ausführung von API Calls

Übung 2

Kanye REST:

Schreibt eine Website, die in der Lage ist, zufällige Zitate von Kanye West zu generieren. Dazu kann die API https://api.kanye.rest/ benutzt werden. Bei jedem Zitat soll es die Möglichkeit geben das Zitat per E-Mail an jemanden zu schicken.