

# 5. Dynamische Webseiten (JavaScript, DOM, BOM)

Prof. Dr. Jürgen Schneider

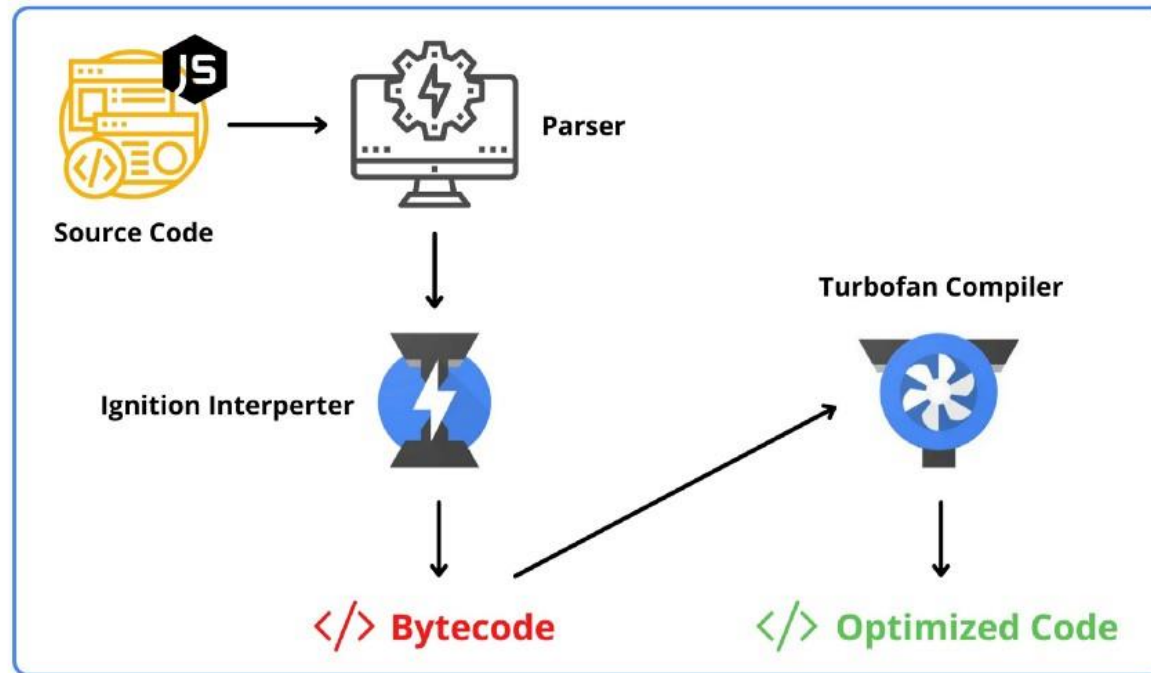
## Lernziele:

- ❑ Einbindung von JavaScript in HTML.
- ❑ Variablen, Konstanten, Datentypen, Operatoren und Funktionen in JavaScript.
- ❑ Arrays, Objekte, Klassen und JSON in JavaScript.
- ❑ DOM-Modell verstehen und anwenden können.
- ❑ BOM-Modell verstehen und anwenden können.

## Überblick:

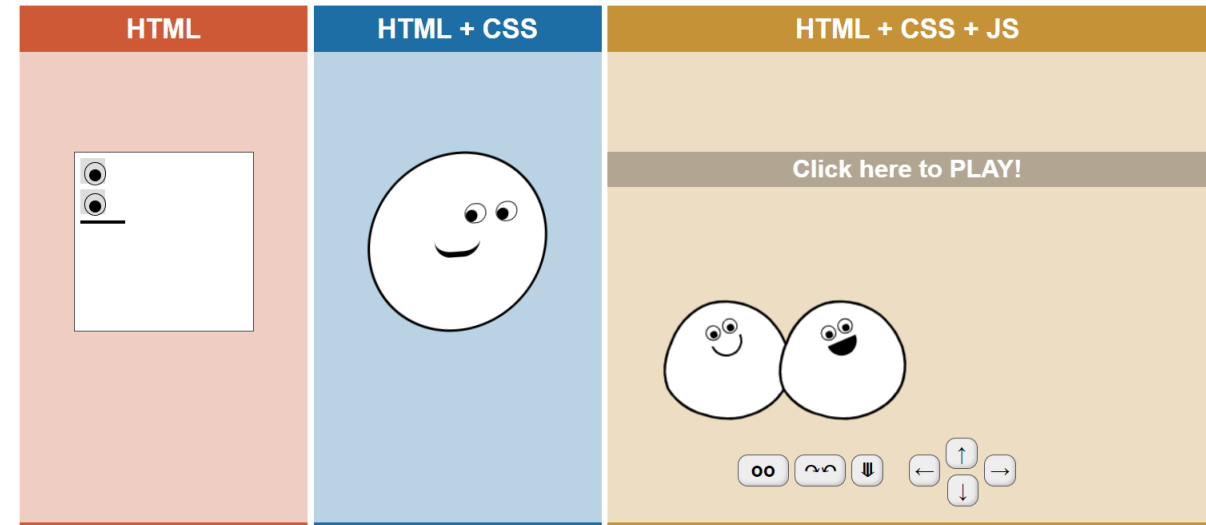
- 5.1 JavaScript – Grundlagen
- 5.2 Arrays, Objekte, JSON und Klassen
- 5.3 Kontrollstrukturen
- 5.4 Funktionen und Best Coding Practices
- 5.5 Document Object Model (DOM)
- 5.6 Browser Object Model (BOM)

## 5.1 JavaScript - Grundlagen



# Separation of concerns

- Gemäß dem Architekturkonzept "Separation of Concerns" haben wir uns zuerst mit
  - **HTML:** und mit
  - **CSS:** Definition der Gestalt einer Webseite beschäftigt.
- Jetzt ergänzen wir die Webseite mit dynamischen Inhalten, also Inhalten, mit denen der Benutzer in der Lage ist mit der Webanwendung zu interagieren. Dazu verwenden wir die Skriptsprache **JavaScript**.
  - **JavaScript:** Dynamische Interaktion mit dem Benutzer.  
Struktur, Inhalt und Gestalt ist über das **DOM-Interface** per JavaScript veränderbar.



Definition der  
Struktur und  
des Inhaltes  
einer Webseite.

Definition der  
Gestalt einer  
Webseite

Dynamische Interaktionen

# JavaScript und ECMAScript

- ❑ **JavaScript** (kurz **JS**) wurde 1995 von **Brendan Eich** entwickelt (Mitarbeiter bei **Netscape**).
- ❑ JavaScript ist eine **Skriptsprache**, die ihren Namen der damaligen **Popularität** von **Java** verdankt (ursprünglich: **LiveScript**).
- ❑ JavaScript besitzt eine **Java-ähnliche Syntax** ist aber von Java grundverschieden.
- ❑ 1996 wurde auf Basis von JavaScript der **ECMAScript – Standard** (ECMA-262) von der ECMA International Organisation spezifiziert und **1997 veröffentlicht**.
- ❑ **2009: JS** ist **Trademark** der Firma Oracle
- ❑ Weitere auf dem ECMA-Standard basierenden Programmiersprachen sind
  - **JScript**: Entwicklung von Microsoft
  - **TypeScript**: Entwicklung von Microsoft

[Back to the list](#)

## ECMA-262

ECMAScript® 2021 language specification

12th edition, June 2021

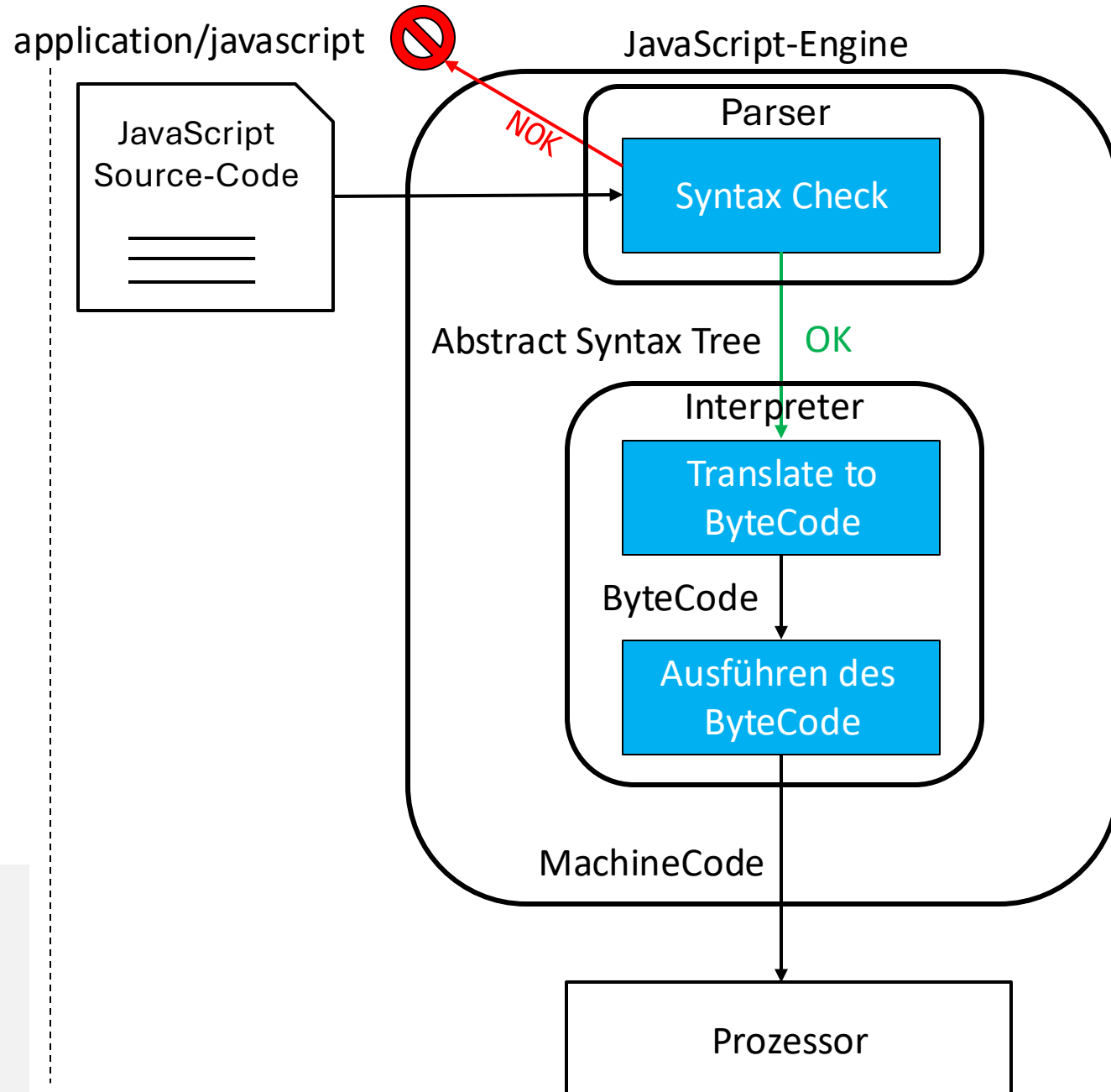
[Link to ECMA-262](#)

**ECMA**: European Computer Manufacturers Association

# JavaScript-Engine

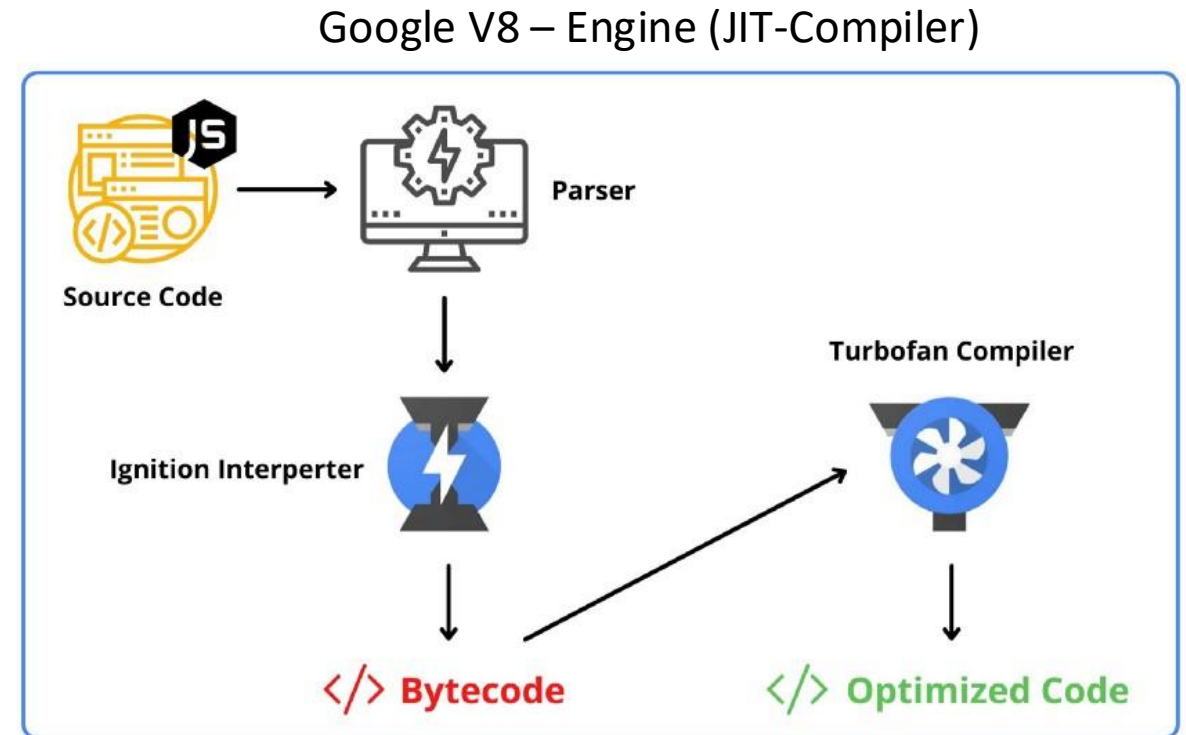
- ❑ JavaScript benötigt eine sogenannte **JavaScript-Engine** zur Ausführung des Quellcodes.
- ❑ Die JavaScript-Engine verarbeitet den Source-Code in mehreren Teilschritten:
  - **Parser**: Liest den Quellcode Zeile um Zeile und überprüft diesen gegen die **JavaScript-Spezifikation**.
    - a. **Fehler**: Error und Stop Execution
    - b. **Erfolg**: Erstellung einer abstrakten Datenstruktur (Abstract Syntax Tree AST)
  - **Interpreter**: Umwandlung des AST in **Byte Code** und anschließende Ausführung des Byte Codes durch den Interpreter.

**Skriptsprachen** (z. B. JavaScript, Python, Bash) werden interpretiert, d. h. der Code wird zur Laufzeit Zeile für Zeile von einem Interpreter ausgeführt, ohne dass ein separater Kompilierungsschritt erforderlich ist.

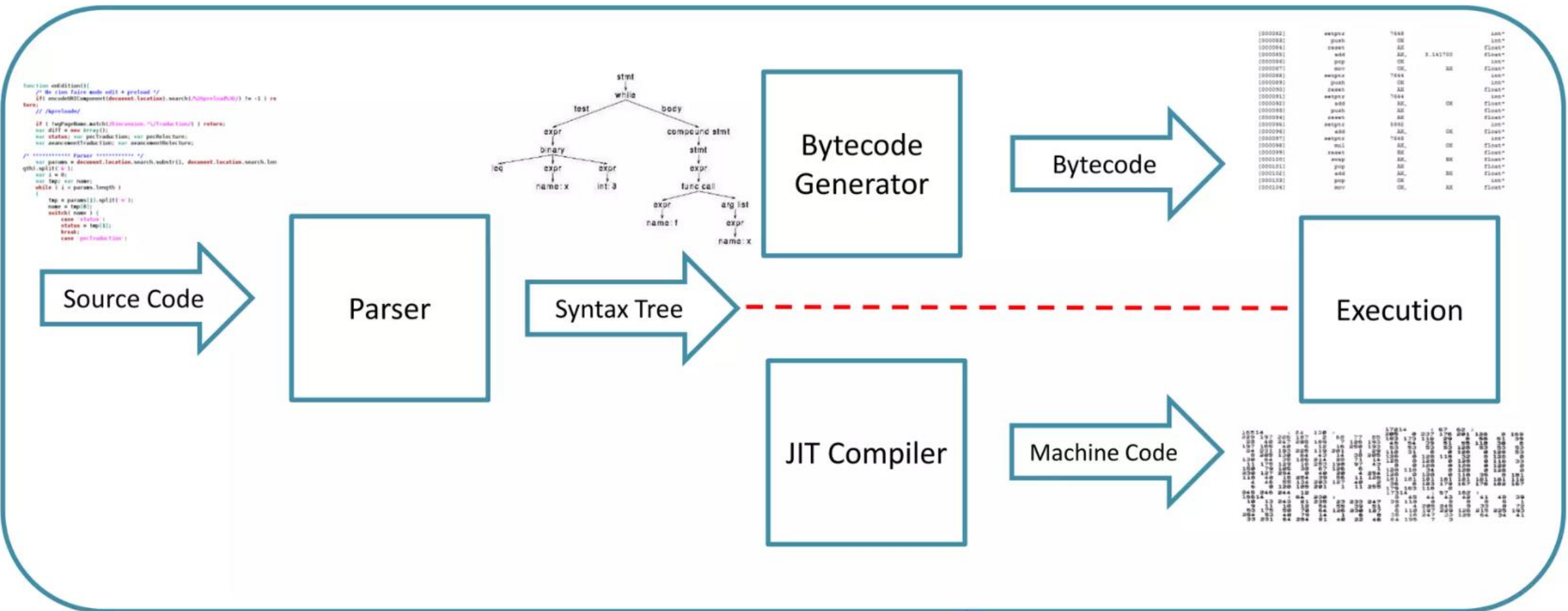


# JavaScript-Engine und JIT-Compiler

- ❑ Um die Performance zu steigern, gibt es mittlerweile JavaScript-Engines mit sogenannten **JIT-Compilern** (Just-In-Time).
  - Interpretieren den Source Code bei der initialen Ausführung.
  - Code-Bestandteile die mehrfach ausgeführt werden (z.B.: Schleife, Funktionen) werden über einen separaten Compiler in **Maschinencode** kompiliert.
- ❑ **Google V8 – Engine**
  - **Parzen** des JavaScript Code und Aufbau eines **Abstract Syntax Tree**.
  - Interpretieren des Codes im **Ignition Interpreter** und Generierung von **ByteCode**.
  - Compilieren von „**Hot**“-Code im **Turbofan** JIT Compiler in **Machine Code**.



# JavaScript-Engine und JIT-Compiler



# JavaScript: Client- und Server-Side

- ❑ Jeder **Webbrowser** besitzt eine **embedded JavaScript-Engine**.
- ❑ Die JavaScript-Engines **unterscheiden** sich zwischen den einzelnen Browsern:
  - Chrome-Browser: **Google V8-Engine**
  - Microsoft Edge: **Google V8-Engine**
  - Firefox-Browser: **SpiderMonkey**
  - Safari-Browser: **JavaScriptCore, Nitro**
- ❑ Es gibt auch **JavaScript-Engines** für **Web-Server**, sodass JavaScript auch für serverseitiges Programmieren von Web-Applikationen eingesetzt werden kann:
  - **node.js** : hat die **Google V8 Engine** integriert

- ❑ **JavaScript** ist **standardmäßig** in allen Webbrowsern installiert und **aktiviert**.



# Integration von JS in HTML

- ❑ In HTML kann JavaScript-Code direkt im Dokument ([intern](#)) zwischen den Tags `<script>` und `</script>` eingefügt.
- ❑ Gemäß dem Konzept [Trennung der Zuständigkeiten](#) und der Idee der [Modularität](#) und [Wiederverwendbarkeit](#), ist es effektiver JavaScript in [externen Dateien](#) bereitzustellen. Dazu verwendet man das `src`-Attribut des `<script>`-Tags.

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

- ❑ JavaScript-Dateien enden mit `*.js`
- ❑ MIME-Type: `application/javascript`
- ❑ Skripte können im `<body>`- und im `<head>`-Bereich einer HTML-Seite oder in beiden platziert werden.

```
<body>
  <h1> Mein Titel </h1>
  <p> JavaScript wird innerhalb script-Tag
    gespeichert ... </p>

  ...

  <script>
    function myFunction(){
      alert( 'Hello, world!' );
    }
  </script>
</body>
```

```
<body>

  <h1> Mein Titel </h1>

  <p> JavaScript wird in einer externen Datei
    gespeichert ... </p>

  <script src="script/myScript.js"></script>

</body>
```

# Interaktive Ausgabe von Daten

- ❑ JavaScript kann Daten auf unterschiedliche Weise interaktiv ausgeben.
- ❑ Daten können in einem HTML-Element ausgegeben werden. Über `document.getElementById()` wird auf das Element zugegriffen und mittels `innerHTML` oder `textContent` ein Inhalt in das selektierte Element geschrieben.
- ❑ Über eine Alert-Box `alert()` kann eine Information in einem `separaten Dialogfenster` innerhalb des `Browserfensters` ausgegeben werden.
- ❑ Während der Programmentwicklung (`Debugging`) kann mittels `console.log()` in die Konsole des Browsers geschrieben werden.

DOM (Document Object Model): `document` – Objekt

```
<p id="p--innerHTML"></p> <p id="p--innerText"></p>
<script>
  document.getElementById("p--innerHTML").innerHTML=
    "HTML als Inhalt ...";
  document.getElementById("p--innerText").textContent=
    "Nur Text als Inhalt ...";
</script>
```

BOM (Browser Object Model): `console.log()`, `alert()`

```
<script>
  console.log("DEBUGGING Output: direkte Ausgabe
    in die Konsole des Browsers.");
  alert('Ausgabe im Browser-Fenster!');
</script>
```

# Interaktive Ausgabe von Daten

- ❑ Vorzeitige Einbettung eines JS-Skriptes macht dann Sinn, wenn das Skript genau an dieser Stelle in der HTML-Seite ausgeführt werden muss.
- ❑ Zum Beispiel soll die Ausgabe einer Meldung genau an dieser Stelle erfolgen:

Webseiten-  
Objekt

Member-Operator

```
document.write('<h3 style="color:red">Ausgabe  
aus einem JavaScript</h3>')
```

Methode schreibt auf Webseite "an der Stelle" des JS-Skripts.

```
<body>  
  <h1> Mein Titel </h1>  
  <p> JavaScript wird sequentiell ausgeführt</p>  
  <p>Hier beginnt JavaScript- Code:</p>  
  <script>  
    document.write('<h3 style="color:red">Ausgabe von  
      einem JavaScript</h3>')  
  </script>  
  <p>Hier endet JavaScript-Code!</p>  
</body>  
</html>
```

## Mein Titel

JavaScript wird sequentiell ausgeführt

Hier beginnt JavaScript- Code:

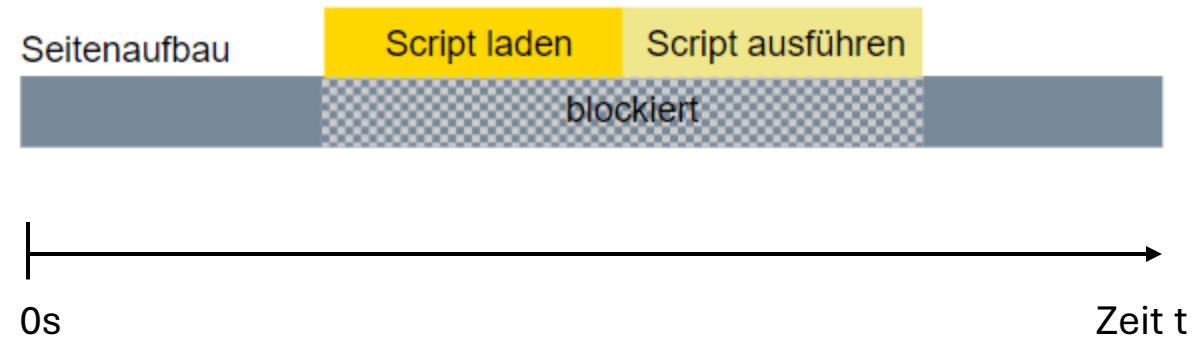
**Ausgabe von einem JavaScript**

Hier endet JavaScript- Code!

# Skriptausführung: Sequentiell

- ❑ Wenn der Browser eine HTML-Datei liest und ein `<script>`-Tag erkennt, lädt der HTML-Parser das Skript und führt es **unmittelbar** aus.
- ❑ Dies wird auch als **sequentielle** Ausführung bezeichnet.
- ❑ **Probleme:**
  - Das **Laden** der Webseite wird **verlangsamt**, da der HTML-Parser den Seitenaufbau stoppt und zuerst das JavaScript laden und dieses dann ausführt.
  - Der DOM-Tree ist ev. nicht vollständig vorhanden, sodass Teile des JS-Skripts nicht funktionsfähig sind.
- ❑ **Best-Practise:**
  - Platzierung des JS-Codes am Ende des HTML-Dokumentes unmittelbar vor dem `</body>`-Tag.

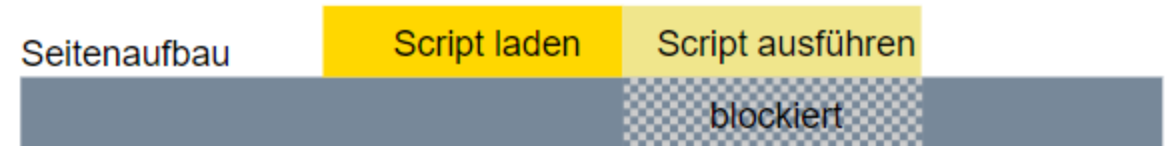
```
<script src="script/myScript.js"></script>
```



# Skriptausführung: Asynchrone Skriptausführung

- Weitere Möglichkeiten JS-Skripte in HTML einzubinden:
  - Verwenden des **async**-Attributs. Der HTML-Parser lädt das Skript asynchron und führt es erst aus wenn das Skript vollständig geladen wurde.  
Mit **async** führt der Browser Skripte in der Reihenfolge »**zuerst geladen zuerst ausgeführt**« aus. Die Reihenfolge im HTML-Code spielt keine Rolle.
  - Verwenden des **defer**-Attributs. Der HTML-Parser lädt das Skript asynchron und **führt es am Ende** nach dem Seitenaufbau aus.  
Mit **defer** führt der Browser **Skripte in der Reihenfolge** aus, in der sie im HTML-Code aufgeführt sind.

```
<script async src="script/myScript.js"></script>
```



```
<script defer src="script/myScript.js"></script>
```



# Skriptausführung: Eventgesteuert

- ❑ Ein Browser ermöglicht die **Interaktion** mit einem Benutzer.
- ❑ Klickt beispielsweise ein Benutzer auf den Button eines Formulars, **feuert** ("fired") das Browserfenster ("window") ein **Event**.
- ❑ Das Event enthält die **Beschreibung** der Benutzerinteraktion: "Maus-Klick auf Button XYZ".
- ❑ Mittels dieses Events kann dann ein spezifischer JavaScript-Code **aufgerufen** ("getriggert") werden.
- ❑ **Beispiel:** Nebestehender Source-Code
  - **onclick**="myMessage()": Aufruf der JavaScript-Funktion **myMessage()**.
  - `<button type="button" ...>` hat kein Default-Verhalten.

```
<body>

  <button type="button" onclick="myMessage()">Start a
      message box.

</button>

<script src="script/myScript.js"></script>

</body>
```

myScript.js

```
function myMessage(){
    alert( 'Hello, world!' );
}
```

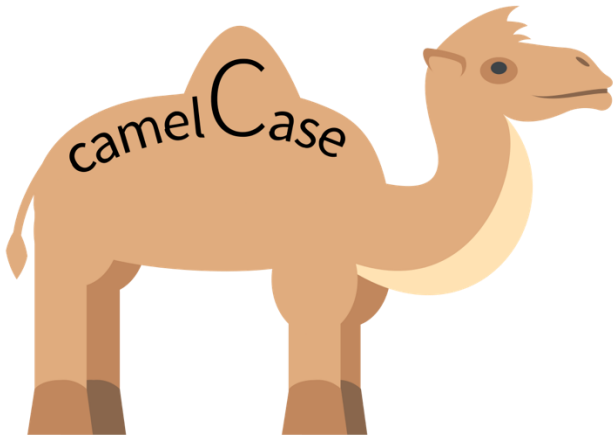
# Kommentare und Bezeichner (Identifier)

- ❑ JavaScript **Kommentare** können analog zur Programmiersprache Java oder C/C++ im Programmcode gekennzeichnet werden mittels

// - Zeilenkommentare

/\* ... \*/ - Blockkommentare

- ❑ Alle JavaScript – Bezeichner (Variablen, Funktionen, Keywords...) sind **case sensitive**, d.h. Groß- und Kleinschreibung wird unterschieden.



- ❑ In JavaScript dürfen **Unicode-Zeichen** für die Bezeichner von Variablen (**Identifier**) verwendet werden.
- ❑ Für die **Identifier** hat sich in JavaScript die **Lower Camel Case** Syntax durchgesetzt:
  - **erster Buchstabe klein**, jedes **neue Wort** beginnt mit einem **Großbuchstaben**
  - `firstName, masterCard, ...`
  - müssen mit einem Buchstaben (oder `_`, `$`) beginnen  
`_lastName, $masterCard`
- ❑ **Best Practise:** Verwenden Sie für Menschen gut lesbare und verständliche Namen, die aus beschreibenden Worten bestehen  
`userName, firstName, age, shoppingCart`

# JavaScript Statements

- Eine JavaScript besteht aus einer Abfolge von **einzelnen Anweisungen** für die JavaScript-Engine.
- Ein **JS-Anweisung** wird mit einem **Semicolon** abgeschlossen

```
let x = 10;  
document.getElementById("p1").textContent=  
    "InnerHTML Output";
```

- JavaScript **ignoriert** Leerzeichen und Zeilenumbrüche.
- JavaScript-Statements können in sogenannten **Code-Blöcken** zusammengefasst werden  
**{ ... }**

Am Ende eines Code-Blockes wird **kein Semikolon** verwendet.

- Beispiele für **Code-Blöcke** sind **Funktionen**, die eine bestimmte Aufgabe erledigen

```
function multiply(x,y) {  
    let c = x * y ;  
    return c;  
}
```

- Oder **Kontrollanweisungen**, die auf Basis von Variablen eine Entscheidung treffen

```
if (hour < 8 || hour > 18) {  
    document.write('Our shop is closed');  
} else {  
    document.write('Welcome');  
}
```



# JavaScript: Variablen und Konstanten

- ❑ Variablen werden mit dem Schlüsselwort **let** **deklariert** (informiert Interpreter über Existenz) und mit "=" **definiert** (Interpreter reserviert Speicherplatz).

```
let x;           // Deklaration einer undefined Variablen
let strWert      = 'Alice';  // Definition String Data Type
let quantity     = 10.5;     // Definiton Numeric Data Type
let inStock      = true;     //Def. Boolean Data Type
let dataObject   = null;     // Def. leere Variable (0x00000000)
```

## Identifier

- ❑ Java verwendet die sogenannte **implizite Deklaration**, der Datentyp (**Number**, **String**, **Boolean**, **Null**) wird bei der Wertzuweisung festgelegt und muss nicht explizit angegeben werden.
- ❑ Variable ohne Wertzuweisung sind vom Datentyp **Undefined**.

- ❑ Zusätzlich gilt in JavaScript **dynamische Typisierung**, sie können eine Variable zuerst als Zahl und dann als String verwenden:

```
let x = 10;
x = 'zehn';
```

- ❑ Deklaration und Wertzuweisung mittels eines Statements

```
let price = 10.0;
let quantity = 5;
let total = price * quantity;
```

- ❑ Mehrfache Deklaration & Wertzuweisung

```
let price = 10.0, quantity=5, total;
total = price * quantity;
```

# JavaScript: Literale

- Ein **Literal** ist die konkrete **Darstellung** eines **Werts**.
  - **Zeichenliterale** (**Datentyp String**) müssen in einfaches oder doppeltes Anführungszeichen gesetzt werden.  
`"Das ist ein Auto."`  
`'Das ist ein Auto.'`
  - **Zahlen** (**Datentyp Number**): Integer, Dezimalzeichen, ein Plus- oder Minuszeichen, Exponent  
Integer: `4711` , `3`  
Negative Ganzzahlen: `-56`  
Floating: `6.29` , `-5.42`  
Exponent: `5e2` ( $5 \cdot 10^2$ ) , `3e-2` ( $3 \cdot 10^{-2}$ )  
Hexadezimal: `0x46` (70) , `0x348` (800)
  - **Wahrheitswerte** (**Datentyp Boolean**):  
`true` , `false`

- Absichtliches Fehlen eines Wertes (**Datentyp Null**)
  - Der Nullwert `null` repräsentiert das **absichtliche Fehlen** eines **Objektwerts**. Der Nullwert wird in booleschen Ausdrücken als **falsch** behandelt.

```
let x = null;  
if (x) {console.log('x is not null')}  
else {console.log('x is null')}
```

- Deklarierte, aber nicht initialisierte Variable hat den **Datentyp Undefined** und den Wert **Undefined**.

```
let x;
```

# JavaScript: Konstanten

- ❑ Mittels `const` deklarierte Variablen besitzen einen **konstanten Wert** für **primitive Datentypen** (Number, String, Boolean) und sind nicht mehr änderbar:

```
const PI = 3.1415;
```

- ❑ **Konvention:** Konstanten, die zu Beginn eines Programmes definiert werden, werden in Großbuchstaben geschrieben, ansonsten wie jede normale Variable (CamelCase):

```
const PI = 3.141;  
  
function perimeter() {  
  // ...perimeter for circle with r = 1  
  const perimeter = 2 * PI;  
  alert(perimeter);  
}
```

- ❑ `const` wird auch verwendet, um **Arrays** und **Objekte** zu deklarieren.
  - In diesem Fall erstellt die `const`-Deklaration einen **nicht mehr änderbaren Verweis (Reference)** auf den **Speicherbereich** des Arrays oder des Objektes im Heap.
  - Der Inhalt eines Arrays oder Objektes kann allerdings verändert werden.

- ❑ JavaScript verwaltet den Speicher mit zwei Bereichen:
  - **Stack (Stapelspeicher)** – Für primitive Werte und Funktionsaufrufe.
  - **Heap (Dynamischer Speicher)** – Für Objekte, Arrays und Funktionen.

# JavaScript: Sichtbarkeit von Variablen und Konstanten

- ❑ **Lokale Variablen:** Mittels `let` und `const` innerhalb eines Code-Blocks `{ ... }` deklarierte Variablen haben nur **lokale Gültigkeit** innerhalb dieses Code-Blocks

```
{
  let a = 2;
  const b = 3;
  let z = a + b //z=5
}
...
{
  let z = a + b;
```

a und b nicht mehr existent

- ❑ Lokale Variablen werden beim Funktionsaufruf im Stack erstellt und nach der Beendigung der Funktion wieder entfernt.

- ❑ **Globale Variablen:** Mittels `let` und `const` außerhalb eines Code-Blocks `{ ... }` deklarierte Variablen sind außerhalb und innerhalb des Blocks gültig

```
let a = 2;
const b = 3;
c = 10;           //globale Variable
{
  let z = a + b;  //a und b noch vorhanden
}
```

- ❑ **Bad Practise:** Deklaration einer Variablen ohne `let`. Wird als globale Variable behandelt.
- ❑ Globale Variablen führen zu **erhöhtem Speicherbedarf** und dem Risiko der **Namenskollision**.

# Assignment & Arithmetische Operatoren

## ❑ Arithmetische Operatoren:

`+` , `-` , `*` , `/` , `**` , `%` , `++` , `--`

```
x = y + z; //Addition (- Subtraktion)
j = ++i;   //Inkrement i by 1 before assignment
j = i--;   //Dekrement i by 1 after assignment
z = 10**5; //10 hoch 5
y = 10 % 3 //Rest (Modulus): 10:3 = Rest(1)
y = 7 * 4  //Multiplikation
x = 10 / 2  //Division
```

## ❑ Assignment Operatoren

`=` , `+=` , `*=` , `/=` , `**=` , `%=`

```
x = 10; //x = 10
y = 9;

x += y; // x= x + y = 19
x *= y; // x= x * y = 90
x /= y; // x= x / y = 0.9
x %= y; // x= x % y Rest(9:10) = 9
x **= y; // x=x**y = 3486784401;
```

# String Operatoren

- ❑ **String Additionen/Verkettung:** Strings und Zahlen

```
//name: "Alice Bob"
name= "Alice" + ' ' + "Bob";
//x: "Alice5"
x = "Alice" + 5;
```

- ❑ Bestimmen der **String-Länge**


```
name= "Alice" ;
y= name.length; //Wert ist 5
```

- ❑ Substring eines Strings (Schneiden):

```
name= "Juergen Schneider";
vorName=name.substring(0,7); //Juergen
```

von  
(included)

bis  
(excluded)



- ❑ Entfernen von Leerzeichen am Anfang und am Ende eines Strings

```
str = ' Juergen Schneider ' ;
name = str.trim(); // 'Juergen Schneider'
```

- ❑ Umwandlung Groß- und Kleinschreibung

```
str = "Mein neues Auto";
str.toLowerCase(); // mein neues auto
str.toUpperCase(); // MEIN NEUES AUTO
```

- ❑ Zerlegen eines Strings anhand eines frei definierbaren Trennzeichens in eine **Liste (Array)**

```
string = "Nagel,Schraube,Dübel"
list = string.split(",");
//[ "Nagel", "Schraube", "Dübel" ]
```

# Comparison Operatoren

## Beispiele für Vergleichoperatoren:

```
let x=10, y = null;
if (x == "10") { ... };    //true, x zu String konv.
if (x === "10") { ... };  //false, different types
if (x !== "10") {... };   //true
if (x != "10") {... };    //false
if (x < "12") {... };     //true
let check = (x > 5);       //check = true
let check = (y === null)  //check = true
```

## Ternary Operator

```
                                true    false
variablename = (condition) ? value1 : value2

voteable = (age < 18) ? "Too young" : "Old
enough";
```

## Liste der Vergleichsoperatoren

Operator	Description
==	Nur Daten werden auf Gleichheit geprüft
===	Daten und Datentyp müssen übereinstimmen
!=	Nur Daten werden auf Ungleichheit geprüft
!==	Daten und Datentyp werden auf Ungleichheit geprüft.
>	Daten sind größer als
<	Daten sind kleiner als
>=	Daten sind größer gleich
<=	Daten sind kleiner gleich
?	Ternary Operator besitzt 3 Argumente

# Logical Operatoren

## □ Logical Operator

```
let x = 6, y=3;
let check;
//check = true
let check = (x < 10 && y > 1);

let x = 5, y=4;
let check;
//check = false
check = !(x == 5 || y == 5)
```

## Logical Operators

Operator	Description
&&	Logisches UND: beide Ausdrücke sind wahr
	Logisches ODER: ein der beiden Ausdrücke ist wahr
!	Logisches NICHT: verändert einen Wahrheitswert !true wird zu false !false wird zu true



# Automatische Typkonvertierungen

- Automatische Typ-Konvertierungen in JavaScript führt teilweise zu **unerwarteten** Ergebnissen

```
x = 5 + null    // x=5      : null konvertiert zu 0
x = "5" + null  // x="5null" : null konvertiert zu "null"
x = "4" + 10    // x="410"   : 10 konvertiert zu "10" -> String-Verkettung
x = "5" + 2     // x="52"    : 2 konvertiert zu "2", da '+' auch für String-Verkettung verwendet wird.
x = "5" - 2     // x=3       : "5" konvertiert zu 5, da '-' nur für die Subtraktion verwendet wird.
x = "5" * "2"   // x=10      : "5" und "2" konvertiert zu 5 und 2 , da '*' für Multiplikation ...
x = "10" / "2"  // x=5       : "10" und "2" konvertiert zu 10 und 2 , da '/' für Division ...
```

# JavaScript: Typeof-Operator

- Der `typeof`-Operator gibt eine **Zeichenfolge** zurück, die den Datentyp einer **Variablen** enthält.

```
//Undefined Variable  
let x ;  
  
//check1 == true, Wert von x ist undefined  
let check1 = (x === undefined) ;  
  
//check2 == true, Typ von x ist undefined  
let check2 = (typeof x === 'undefined');
```

## 5.2 Arrays, Objekte, JSON und Klassen

```
class Person {  
    constructor(vName, nName) {  
        this.vName = vName;  
        this.nName = nName;  
    }  
}
```

# JavaScript: Arrays

- ❑ JavaScript - Arrays sind Listen, die Werte **unterschiedlichen Typs** enthalten können.
- ❑ Einen neuen **leeren Array** erstellt man mit

```
const arr = [ ];
```

- ❑ Eine **Initialisierung** eines Arrays erfolgt durch die direkte Angabe der Werte

```
const arr = ["Leiter", 45, "Personen"];
```

- ❑ Auf Elemente eines Arrays kann über deren **Indexnummer** zugegriffen werden, die Indexnummer **startet** immer bei 0.

```
let str = arr[0]; //str hat Wert Leiter  
let i   = arr[1]; //i hat den Wert 45
```

- ❑ **Verändern** von Werten eines Arrays

```
//Ersetzen von "Personen" durch "Bob"  
arr[2] = "Bob"
```

# JavaScript: Arrays

- ❑ Ein Array stellt neben seinen **Werten** auch **Methoden (Funktionen)** zum Zugriff auf die Werte bereit.
- ❑ Auf die Variablen und Methoden kann über die **Punktnotation** zugegriffen werden.
- ❑ Mittels **push(Wert)** können neue Werte am **Ende** der **Liste** hinzugefügt werden, mittels **pop()** werden diese entfernt:

```
const arr = ["Leiter", 45, "Bob"];
arr.push(99);
// output: Array ["Leiter", 45, "Bob", 99]
console.log(arr);
let x= arr.pop();
//output: x=99, array ["Leiter", 45, "Bob"]
console.log(x, arr);
```

- ❑ Mittels **unshift()** kann ein neuer Wert am **Anfang** einer Liste **hinzufügt** werden, mittels **shift(Wert)** wird ein Wert am **Anfang** einer Liste **entfernt**:

```
//Lesen und Entfernen des 1. Wertes
const arr = [1, "Zwei", 3];
let firstElement = arr.shift();
console.log(arr); // output: ['Zwei',3]
console.log(firstElement); // output: 1

//Hinzufügen von "blue" an die 1. Stelle
arr.unshift("blue");
console.log(arr); // output: ['blue','Zwei',3]
```

# JavaScript: Arrays

- Die Länge eines Arrays erhält man über die `length`-Variable:

```
arr = ["Leiter", 45, "Personen"];

console.log(arr.length); // Länge ist 3
```

- Mittels der `sort()`-Funktion (`reverse()`) lässt sich ein Array **lexikografisch** (nach **Unicode-Wert**) sortieren:

```
fruits = ["Ban", "Ora", "Apple", "Man"];
fruits.sort();
console.log(fruits);

//output: ["Apple", "Ban", "Man", "Ora"]
```

- Um **Zahlen zu sortieren**, benötigt man eine Vergleichsfunktion, da ansonsten beispielsweise die Zahl 100 wegen der führenden "1" vor der Zahl "40" mit der führenden "4" gelistet wird.

```
points = [40, 100, 1, 5, 25];
points.sort();
console.log(points);
//[1 , 100, 25, 40, 5]
```

- Elementweise Vergleichsfunktion in `sort`-Methode:
  - negativer Wert: **a vor b** sortieren
  - positiver Wert: **b vor a** sortieren

```
function(a, b){return a - b}
```

```
points = [40, 100, 1, 5, 25];
points.sort(function(a, b){return a - b});
console.log(points);

//output: [1 , 5, 25, 40, 100]
```

# Assoziative Arrays / Objekte

- ❑ Bei einem Array konnten Sie über die Indexnummer auf ein Array-Element direkt zugreifen.
- ❑ Wenn Sie auf die Werte eines Arrays über sprechende **Namen** oder besser über **Wörter** zugreifen möchten, stellt Ihnen JavaScript ein sogenanntes **assoziatives Array** zur Verfügung (andere Namen: **Dictionary**, **assoziativer Speicher**).
- ❑ In JavaScript werden **assoziative Arrays** auch als **Objekte** bezeichnet.
- ❑ Ein assoziatives Array wird in geschwungene Klammern in Form von **Name-Wert** Paaren angegeben.

```
{"Name1": Wert1, "Name2": Wert2, ... }
```

- Die Anführungszeichen (" " oder ' ') sind nicht zwingend erforderlich, wenn der Schlüssel ein gültiger Bezeichner (Buchstaben) in JavaScript ist.

- ❑ Die **Name-Wert-Paare** in einem Objekt werden auch als **Eigenschaften** bezeichnet und können **Zahlen**, **Strings**, **Boolesche Werte**, **Arrays**, **Objekte** oder **Funktionen** enthalten.
- ❑ Initialisierung eines assoziativen Arrays

```
const phoneBook = { };
```

- ❑ Klassisches Beispiel: Abspeichern eines Telefonbuches

```
const phoneBook = {"Alice" : 5734, "Bob":  
1345, "Sally": 6352};
```

# Assoziative Arrays / Objekte

- ❑ Zugriff auf den Wert eines Arrays erfolgt dann über die Angabe des Namens (en: **Property**) in **Punktnotation** oder in **[ ]-Klammern**

- ❑ Hinzufügen eines neuen Eintrages am Ende des Arrays erfolgt über

```
phoneBook["Klaus"] = 3976;  
phoneBook.Klaus = 3976;
```

- ❑ Ändern eines vorhandenen Eintrages

```
phoneBook.Bob = 5634;  
phoneBook["Bob"] = 5634
```

- ❑ Das Löschen eines Eintrages mittels der delete Methode

```
delete phoneBook["Sally"];  
delete phoneBook.Sally;
```



# Assoziative Arrays / Objekte

- ❑ Wird eine **Funktion** in einem Objekt gespeichert, wird Sie auch als eine **Methode** des Objektes bezeichnet.
- ❑ Beispiel: Objekt **objJohn** einer Person

```
const objJohn = {  
  firstName: "John",  
  lastName : "Doe",  
  id       : 5566,  
  fullName : function(title) {  
    return title + " " + this.firstName  
    + " " + this.lastName;  
  }  
};
```

- ❑ Schlüsselwort **this** referenziert das aufrufende Objekt selbst.

- ❑ Im Beispiel ist es das **Objekt objJohn**, das eine Methode **fullName** besitzt mit den folgenden Eigenschaften:
  - **this.firstName**: Wert ("John") von der Property **firstName**
  - **this.lastName**: Wert ("Doe") von der Property **lastName**
  - **return** : "John Doe"
- ❑ Die **Methode** wird über die **Punktnotation** aufgerufen

```
name = objJohn.fullName("Dr.");  
//name= "John Doe"
```

# Assoziative Arrays / Objekte

- Wenn Sie die **Methoden-Property**, also den **Namen** der Methode **ohne die ()** – Klammern aufrufen, wird der Methoden-Quellcode also der Property-Wert der Funktion ausgegeben

```
name = objJohn.fullName;  
//name = "function () {...}"
```

# JSON

- ❑ JSON ist die Abkürzung für **J**ava**S**cript **O**bject **N**otation.
- ❑ JSON wurde initial für JavaScript entwickelt, stellt aber mittlerweile einen weitverbreiteten Standard dar, um **Daten** im **WWW** zwischen **Client** und **Server** auszutauschen.
- ❑ JSON wird dazu verwendet ein JavaScript Objekt zu **serialisieren**, d.h. das Objekt in einen **JSON-encoded String** umzuwandeln:

```
let jsonString = JSON.stringify(itProBob);
```

1

- ❑ Der Empfänger der serialisierten Daten kann diese dann wieder in ein JavaScript Objekt zurückwandeln

```
const objBob = JSON.parse(jsonString);
```

2

```
itProBob = {  
  name: 'Bob',  
  age: 30,  
  isAdmin: true,  
  skills: ['Unix', 'JS', 'node.js'],  
  department: 'IT-Operation' };
```

JSON-encoded String

```
{"name":"Bob","age":30,"isAdmin":false,"skills":["  
Unix","JS","node.js"],"department":"IT-Operation"}
```

decoded JavaScript Object

```
objBob = {  
  name: "Bob",  
  age: 30,  
  isAdmin: true,  
  skills: [ 'Unix', 'JS', 'node.js' ],  
  department: "IT-Operation"}
```

# Eigenschaften von JSON-Strings

- ❑ Strings in **JSON-encoded** Objekten werden generell mit **doppelten** Anführungsstrichen dargestellt:  
'Bob' → "Bob"
- ❑ Die Property-Namen des ursprünglichen Objektes werden ebenfalls in doppelte Anführungszeichen gesetzt:  
age:30, → "age":30,
- ❑ JSON ist ein **reines JS-Datenobjekt** und unterstützt **keine Methoden**. In einem Objekt vorhandene Methoden werden entfernt.

- ❑ JSON unterstützt auch ineinander verschachtelte Objekte

```
const obj1 = {  
    obj1Property1: value1,  
    obj1Property2: value2,  
    ...,  
    obj2: {  
        obj2Property1: value1,  
        ...,  
    }  
}
```

# JavaScript: Klassen

- ❑ Jedes Objekt mussten Sie bisher explizit mit den Werten seiner Eigenschaften und Methoden anlegen. Dabei können sich schnell Fehler einschleichen.
- ❑ Mit dem Keyword **Class** können Sie eine **Vorlage (Template, Klasse)** für ein **JavaScript-Objekt** erstellen.
- ❑ Eine Klasse enthält **immer** die Methode **constructor( )**.
- ❑ Die **Konstruktormethode** ist eine spezielle Methode, mit den Eigenschaften
  - muss genau den Namen **constructor** haben
  - wird **automatisch ausgeführt**, wenn ein neues Objekt erstellt wird (**new( )** – Operator)
  - **initialisiert** die Properties.
- ❑ Konvention:
  - Klassennamen erster Buchstabe groß
  - Rest CamelCase

## ❑ Beispiel: **class Person**

Besitzt die Eigenschaften: Vorname, Nachname

```
class Person {  
    constructor(argVorname, argNachname) {  
        this.firstName = argVorname;  
        this.lastName = argNachname;  
    }  
}
```

- ❑ Ein Objekt wird dann mittels des **new-Operator** und dem Konstruktoraufruf erzeugt

```
const klaus = new Person("Klaus", "Mayr");  
const anton = new Person("Anton", "Huber");  
console.log(klaus.firstName + ' ' +  
            klaus.lastName);
```

# JavaScript: Klassen

- Ein Objekt besitzt nicht nur Eigenschaften sondern auch Methoden. Diese werden bei der Klassendefinition nach dem constructor( ) gelistet:

```
class ClassName {  
  constructor() { ... }  
  method_1() { ... }  
  method_2() { ... }  
  method_3() { ... }  
}
```

- Beispiel: Person

Person besitzt neben einem Vornamen, einem Nachnamen zusätzlich die Eigenschaft Geburtsjahr `birthYear` und die Methode `age()`, die sein aktuelles Alter bestimmt.

```
class Person {  
  constructor(argVorName, argNachName, argbirthYear) {  
    this.firstName = argVorName;  
    this.lastName = argNachName;  
    this.birthYear = argbirthYear;  
  }  
  age ( ) {  
    const date = new Date();    //date - Objekt  
    return date.getFullYear() - this.birthYear;  
  }  
}  
  
const klaus = new Person("Klaus", "Mayr", 1995);  
console.log(klaus.firstName + ' ' +  
            klaus.lastName + 'ist ' + klaus.age());  
//Klaus Mayr ist 27
```

# JavaScript: Vererbung von Klassen

- ❑ Mit dem Schlüsselwort **extends** werden die **Properties und Methoden** einer (übergeordneten) Klasse an eine (untergeordnete) Klasse vererbt.
- ❑ Die **super()**-Methode ruft den **Konstruktor** der übergeordneten Klasse (Elternklasse) und initialisiert die vom Elternelement geerbten Eigenschaften.

```
class itProClass extends Person {
    constructor(job, skills, department,
        argVorname, argNachName, argbirthYear ) {
        super(argVorname, argNachName, argbirthYear);
        this.isAdmin = job;
        this.skills = skills;
        this.department = department
    }

    const objAliceITPro =
        new itProClass('Dev', ['html','js'],
            'IT-DEV', 'Alice'
            'Taylor', 2000);

    //Call extended parent method age()
    objAliceITPro.age();
```

# Zusammenfassung: Datentypen in JavaScript

## ❑ JavaScript kennt die folgenden **Datentypen**

- String: "Mein Name ist ...", '007'
- Number: 10, -5, 5.41, -5e3
- Boolean: true, false
- Array: `const cars = ["VW", "Porsche", "BMW"];`
- Objekte (Assoziatives Arrays):  

```
const person = {  
  "firstName" : "James",  
  "lastName"  : "Bond",  
  "profession" : "secret agent"};
```
- Funktionen: `function add (a,b) {...}`

## ❑ Mittels dem **typeof** Operator kann der Typ einer Variablen bestimmt werden:

```
typeof person; // → object
```

## ❑ Typen-Konvertierungen

- String to Number:  

```
x = Number("3.14") ;    // 3.14  
console.log (+ "3.14") //Unary "+" Operator: 3.14
```
- Number to String:  

```
pi = String(3.14); // "3.14"  
pi = (3.14).toString(); // "3.14"
```
- Converting Boolean  

```
Number(false)    // returns 0  
Number(true)     // returns 1  
String(false)    // returns "false"  
String(true)     // returns "true"
```

Der **unäre Plusoperator** wird vor einer einzigen Operandenangabe verwendet und versucht, den Operanden in eine Zahl umzuwandeln.



## 5.3 Kontrollstrukturen

```
switch(expression) {  
    case x:  
        // code block if x === expression  
        break;  
    case y:  
        // code block y === expression  
        break;  
    default:  
        // default code block  
}
```

# Kontrollstruktur: if

- ❑ JavaScript kennt die folgenden Kontrollstrukturen:
  - Exklusive Bedingungen: if / else if / else
- ❑ Bedingungen:

```
if (condition1) {  
    /* execute if condition1 is true */  
} else if (condition2) {  
    /* execute if the condition1 is false and  
       condition2 is true */  
} else {  
    /* execute if the condition1 and condition2 is  
       false */  
}
```

- ❑ Beispiel:

```
let saldo = haben - soll ;  
  
if (saldo < 0) {  
    message="Transaktion ist nicht erlaubt.";   
} else if (saldo === 0) {  
    message="Ihr Konto steht auf Null.";   
} else {  
    message="Transaktion ist erlaubt.";   
}
```

# Kontrollstruktur: switch

- ❑ Fallunterscheidung mit `switch`:
  - `switch()` - `Ausdruck` wird initial ausgewertet.
  - Wert des `switch()` - `Ausdruck` wird mit den `Werten jedes Falles` verglichen. Bei Übereinstimmung wird der zugehörige Codeblock ausgeführt.
  - Liegt keine Übereinstimmung vor, wird der `default - Block` ausgeführt.
  - Wenn JS auf ein `break-Schlüsselwort` trifft, wird die Ausführung des switch-Blockes an dieser Stelle `beendet`.
  - `switch` verwendet `"==="` – Vergleiche.
  - Das `break-Schlüsselwort` kann auch in `Schleifen` verwendet werden.

```
switch(expression) {  
  case x:  
    // code block if x === expression  
    break;  
  case y:  
    // code block y === expression  
    break;  
  default:  
    // default code block  
}
```

# Beispiel für switch

```
let x = "0";
switch (x) {
  case 0:
    text = "Off";
    break;
  case 1:
    text = "On";
    break;
  default:
    text = "No value found";
}
```

text = ?

text = "No value found";

# Beispiel für Switch

```
const user = {
  firstName: "Paul",
  lastName: "Breitner",
  email: "paul.breitner@fcb.com",
  number: 491111,
};

function checkUser () {
  switch (true) {
    case (user===undefined):
      console.log("User must be defined.");
      break;
    case (user.firstName==="undefined") || (user.lastName ==="undefined"):
      console.log("User's first/last name missing");
      break;
    case (typeof user.firstName !== "string") || (typeof user.lastName !== "string"):
      console.log("User's first/last name must be a string");
      break;
    default:
      return user;
  }
}
```

auf "true" prüfen



# for - Schleife

- ❑ **for-Schleifen** führen einen **Code-Block mehrmals** aus, wobei sich die Werte von bestimmten Variablen jedesmal ändern.
- ❑ Vor allem im Umgang mit **Arrays** sind FOR-Schleifen sehr nützlich.
- ❑ **Klassische for-Loop**: wird eine spezifische Anzahl **n** durchlaufen

```
for (Ausdruck 1; Ausdruck 2; Ausdruck 3) {  
    // code block to be executed  
}
```

```
const x = [1,2,3];  
let n = 3;
```

```
for(let i = 0; i < n ; i++) {  
    x[i] = x[i] * 4;  
    console.log(x[i],i);  
} //x === [4,8,12]
```

i	x[i]
0	4
1	8
2	12

**Ausdruck 1** wird (einmal) vor der Ausführung des Codeblocks ausgeführt.

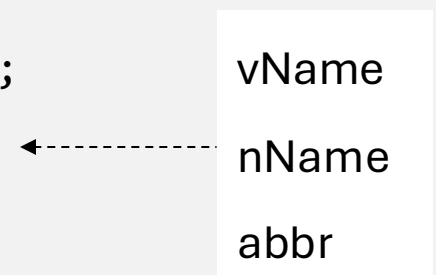
**Ausdruck 2** definiert die Bedingung für die Ausführung des Codeblocks.

**Ausdruck 3** wird (jedes Mal) ausgeführt, nachdem der Codeblock ausgeführt wurde.

# for-in Schleife

- ❑ **for-in-Loop:** Iteration durchläuft die **Indices** bzw. die **Namen eines Arrays** oder die **Properties eines Objektes**.
- ❑ Liefert pro Iteration das zugehörige **Schlüsselwort** oder den zugehörigen **Index** für die **Laufvariable i**.
- ❑ Beispiel:
  - **for-in-Loop** iteriert über die Eigenschaften des Objektes `person`
  - Liefert pro Iteration das **Schlüsselwort** (`vName`, `nName`, `abbr`) der zugehörigen Eigenschaft.

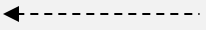
```
const person = {vName:"James", nName:"Bond",  
                abbr:"007"};  
  
let text = "";  
for (let i in person) {  
    text += person[i] + ' ';  
    console.log(i);  
}  
  
//text="James Bond 007"
```



# for-of Schleife

- ❑ **for-of-Loop:** Iteration durchläuft die **Werte** eines **Arrays**.
- ❑ Bei jeder Iteration wird der **Laufvariablen** **x** der Wert der nächsten Eigenschaft zugewiesen.
- ❑ Beispiel:
  - **for-of-Loop** iteriert über die Werte des Objektes `txtBausteine`
  - Liefert pro Iteration den **Wert** ("1", "mal", ...) der zugehörigen Eigenschaft.

```
const txtBausteine = ["1", "mal", "1", "=", "1"];
let message = "";
for (let x of txtBausteine) {
  message += x + ' ';
  console.log(x);
}
//message= "1 mal 1 = 1"
```



1  
mal  
1  
=  
1

kürzer mit **forEach()**-Methode:

```
txtBausteine.forEach(item => console.log(item));
```



# While-Schleife

- Die **while-Schleife** durchläuft einen Codeblock, solange eine **angegebene Bedingung wahr** ist.

```
while (condition) {  
    // code block to be executed  
}
```

- Die **Do-While**-Schleife ist eine Variante der While-Schleife. Der Codeblock wird mindestens einmal ausgeführt, bevor überprüft wird, ob die Bedingung wahr ist und dann wiederholt solange die Bedingung wahr ist.

```
do {  
    // code block to be executed  
}  
while (condition);
```

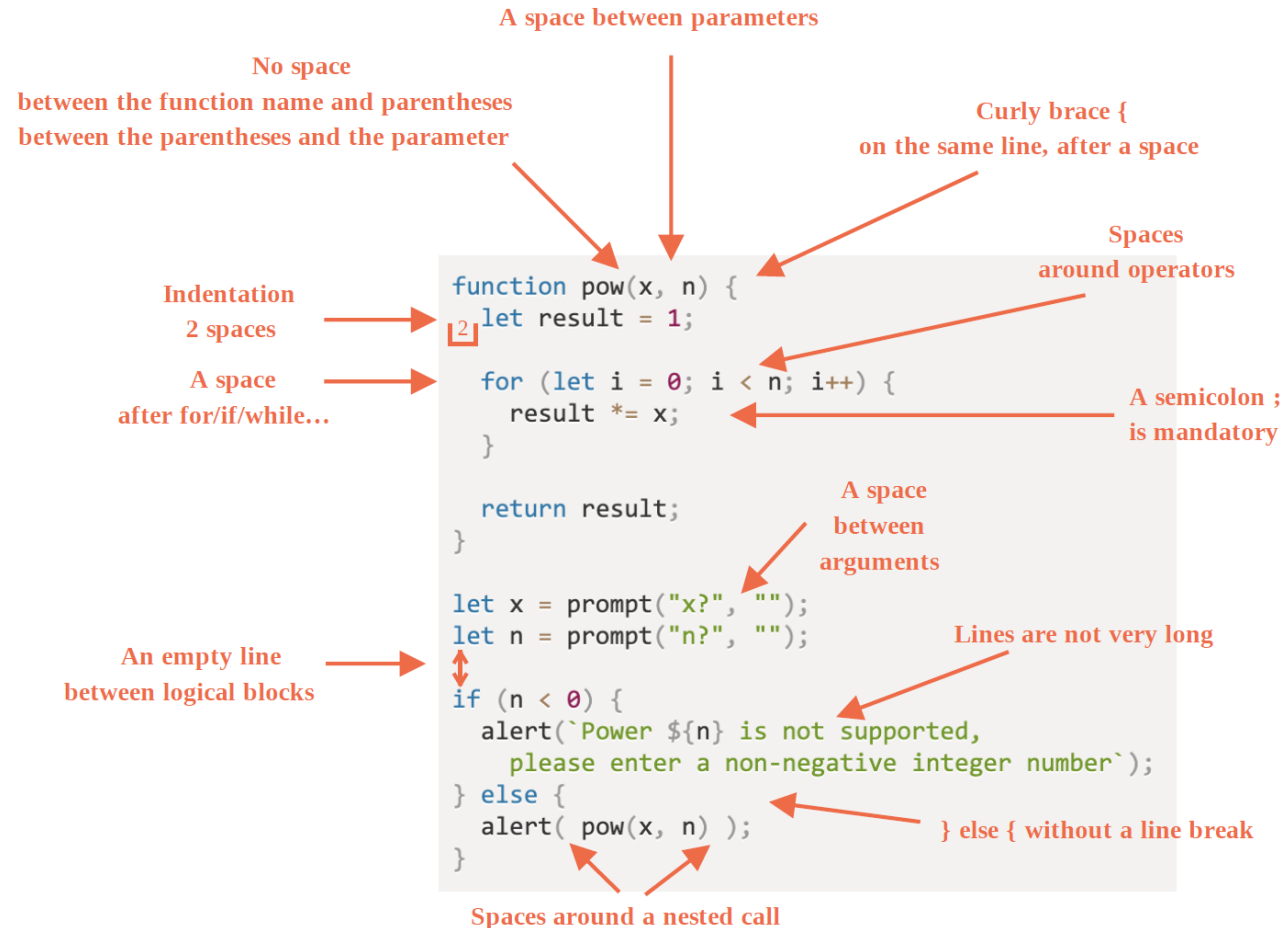
```
const cars = ["Porsche", "Mercedes", "VW", "BMW"];  
let i = 0;  
let text = "";
```

```
while (cars[i]) {  
    text += cars[i];  
    console.log(cars[i]);  
    i++;  
}
```

Porsche  
Mercedes  
VW  
BMW



## 5.4 Funktionen und Best Coding Practices



# Funktionen

- ❑ JavaScript unterstützt die **funktionale Programmierung**, d.h. Funktionen können wie **Variablen** verwendet werden.

**Funktionale Programmierung** erlaubt

- die **Definition** von Funktionen,
- die **Speicherung** und **Verknüpfung** von Funktionen mittels **Variablen**,
- die **Übergabe** von Funktionen als **Parameter** einer anderen Funktion
- oder als **Rückgabewert einer Funktion**.

- ❑ **Funktionen** können mit der Schlüsselwortfunktion **function** gefolgt durch einen **Namen** und dem **Operator ( )** deklariert.

```
function calcSum(a, b) {  
    return a + b;  
}
```

- ❑ Der auszuführende Code wird innerhalb von geschweiften Klammern **{...}** gespeichert.
- ❑ Der **Funktionsoperator (a,b)** enthält die Eingabe-**Parameter** der Funktion.
- ❑ Die Funktion kann über ihren Namen **aufgerufen** werden und wird erst **dann ausgeführt**.
- ❑ Der Rückgabewert wird mittels **return** zurückgegeben.

**Semikolons** werden verwendet, um ausführbare JavaScript-Anweisungen zu trennen.

Da eine **Funktionsdeklaration** keine ausführbare Anweisung ist, wird kein Semikolon am Ende verwendet.

# Funktionen

- Funktionen können wie **Variablen** verwendet werden.

```
function addition (a,b) {return a+b;}

let x = addition(2,3);
let text= "Ergebnis: " + addition(2,3) + " !";
```

- JavaScript bietet **anonyme Funktionen**.
  - Diese besitzen **keinen Namen**, sondern können **direkt** in einer **Variablen** zwischengespeichert werden, um Sie dann **später aufzurufen**:

```
let multiply = function (a, b) {return a * b;};
let z = multiply(2,3); //z = 6
```

- **Immediately-invoked Functions Expression (IIFE)**, sind anonyme Funktionen, die **deklariert** und **sofort ausgeführt** werden.

- function-Deklaration wird durch **()**-Klammern umschlossen.
- Aufruf der Funktion über Operator-Angabe **(2,3)**

```
let area = (function (a, b) {return a * b;}) (2,3);
console.log(area); // 6
```

# Arrow Functions

- ❑ Mittels der Arrow-Syntax  $\Rightarrow$  lassen sich JavaScript-Funktion kompakter schreiben:

- Schlüsselwort `function` entfällt.
- Schlüsselwort `return` entfällt (optional).
- `{}`-Klammern für Code-Block entfallen (optional).

```
(arg1, arg2, ..., argN)  $\Rightarrow$  expression;
```

- ❑ Beispiel:

```
//Ohne Arrow Schreibweise  
let addition = function (a, b) {return a + b};  
  
//Arrow Schreibweise  
let addition = (a, b)  $\Rightarrow$  a + b;  
let addition = (a, b)  $\Rightarrow$  {return a + b};  
  
//Aufruf in beiden Fällen  
let z = addition(2,3);    //5
```

- ❑ Bei nur **einem Argument**, können die `()`-Klammern **weggelassen** werden

```
//Mit ()-Klammern  
let message = (msg)  $\Rightarrow$  alert(msg);  
  
//Ohne ()-Klammern  
let message = msg  $\Rightarrow$  alert(msg);
```

## Immediately-Invoked:

```
let z = ( (a, b)  $\Rightarrow$  a + b ) (3, 4);  
let z = ( (a, b)  $\Rightarrow$  {return a + b;} )(3, 4);
```

# Beispiel: Arrow-Funktion

- ❑ Die BOM-Funktion `prompt()` generiert ein **Eingabefenster** im **Browser**.
- ❑ **Beispiel:** Der vom Benutzer in das Eingabefenster eingegebene Wert wird in einem **Ternary Operator** überprüft. In Abhängigkeit vom Überprüfungsergebnis wird eine von 2 möglichen **Arrow-Funktionen** ausgewählt und diese anschließend ausgeführt.

default

```
let age = prompt("Wie alt sind Sie?", 18);  
let welcome = (age < 30) ?  
    () => alert('Du bist ja noch ein Kind!') :  
    () => alert("Hi, Chef!");  
  
welcome();
```

# Callback - Funktion

Eine **Callback-Funktion** ist eine Funktion, die einer anderen Funktion als **Argument übergeben wird** und später in einem bestimmten Kontext **ausgeführt wird**.

- ❑ Callback-Funktionen werden für die **asynchrone Verarbeitung** von Daten verwendet. Beispielsweise wird eine Callback-Funktion nach dem Laden von Daten via HTTP mittels der **fetch()**-Methode ausgeführt.
- ❑ Callback-Funktionen werden für die **ereignisgesteuerte Verarbeitung** verwendet. Bestimmte Ereignisse (Events) lösen unterschiedliche Funktionsaufrufe aus.
- ❑ **Anonyme/Arrow/Normale - Funktionen** sind das ideale Tool für eine Callback-Funktion.

**//Funktionsdefinition mit Callback-Funktion**

```
function addCallback(a, myfunc) {  
    let b=2;    //local parameter  
    return a + myfunc(a,b);  
}
```

**//Callback mit Funktionsnamen**

```
function calcMultiply(a, b) { return a*b; }  
let zf2 = addCallback(a , calcMultiply);
```

**//Callback mit anonymer Callback-Funktion**

```
let zf1 = addCallback(a ,  
    function (e,f) {return e * f});
```

# Beispiele: Callback-Funktion

- Jede Sekunde soll die aktuelle Uhrzeit in ein `<p>`-Element mit

```
<p id="timer"> </p>
```

auf der Webseite ausgegeben werden.

- Dazu wird der BOM-Funktion `setInterval()` eine **Callback-Funktion** übergeben, die die aktuelle Uhrzeit alle `1000ms` bestimmt und in das HTML schreibt

```
setInterval( () => {  
    const myDate = new Date();  
    document.getElementById("timer").innerHTML=  
        myDate.getHours() + ":"  
        + myDate.getMinutes() + ":"  
        + myDate.getSeconds();  
    }, 1000);
```

- `Date` ist eine in JavaScript **eingebaute Klasse** und speichert die lokale Zeit.



# Eigenschaft von Funktionen

## ❑ Eigenschaften einer Funktion:


- Kann Parameter haben (keine vordefinierten Datentypen).
- Parameter werden in lokale Variablen kopiert.
- Kann einen Rückgabewert haben (kein vordefinierter Rückgabotyp)  
**Best Practise: immer Rückgabewert und diesen prüfen**
- Funktionsaufruf wird nur vom **Funktionsnamen** **abgeleitet**, nicht von Parametern.
- Fehlende Parameter werden durch "undefined" Variablen ersetzt.
- Zusätzlich übergebene Parameter werden ignoriert.

- Der Zugriff auf eine Funktion ohne **()** gibt die Funktionsdefinition anstelle des Funktionsergebnisses zurück.
- Variablen die mit **let, const** innerhalb einer Funktion deklariert, sind lokal und nur innerhalb der Funktion zugreifbar.
- Variablen, die **außerhalb** eines Funktionsblockes definiert wurden, sind innerhalb der Funktion zugreif- und veränderbar.

# JavaScript: Method Chaining

- ❑ **Methodenverkettung (Method Chaining):** Gibt eine Methode das gleiche Objekt (**this**) als Ergebnis zurückgibt, kann man eine weitere Methode des Objektes unmittelbar anwenden.
- ❑ Bei **String-Manipulationen** wird dies häufig angewandt, um sich einerseits Schreibarbeit und andererseits einen übersichtlichen Source-Code zu erstellen,
- ❑ **Beispiel:**
  - (1) **trim()** → Rückgabe **string**: Entfernen von Leerzeichen am Anfang und am Ende
  - (2) **toLowerCase()** → Rückgabe **string**: Groß- zu Kleinbuchstaben konvertieren
  - (3) **split()** → Rückgabe **array**: Zerlegen eines Strings in ein Array → kein weiteres chaining möglich.

Chaining - Direction



```
name = " James, Bond , 007 ";
const list = name.trim().toLowerCase().split(',');
console.log(list[0]+list[1]+list[2]);
// ["james", "bond", "007"]
```

# Coding Practices

Die Kunst des Programmierens besteht darin eine komplexe Aufgabe so zu programmieren, das der Programmcode einen **einfachen und klaren Aufbau** besitzt und von Menschen **gut lesbar** ist.

Zudem sollte der Programmcode die Ressourcen der Zielplattform optimal nutzen.

- ❑ **Modularisierung** des Programmcodes durch den Einsatz von **Funktionen** und **Objekten**.
- ❑ **Variablen-**, **Objekt-** und **Funktionsnamen** sollten eine klare, sich selbstbeschreibende Bedeutung haben.
- ❑ Damit ein **Funktionsname** den **Inhalt** der Funktion beschreiben kann, sollte die Funktion auch nur **eine Aufgabe** erledigen.

Prinzip: **Eine Funktion – Eine Aktion**

fehlende Modularität,  
fehlende Struktur,  
nicht wartungsfähig

zu viele Klassen,  
zuviel an Struktur,  
tiefe Verschachtelung



## Spaghetti code

Unstructured and hard-to-maintain code caused by lack of style rules or volatile requirements. This architecture resembles a tangled pile of spaghetti in a bowl.



## Lasagna code

Source code with overlapping layers, like the stacked design of lasagna. This code structure makes it difficult to change one layer without affecting others.



## Ravioli code

Isolated bits of code that resemble ravioli. These are easy to understand individually but—taken as a group—add to the app's call stack and complexity.



## Pizza code

A codebase with interconnected classes or functions with unclear roles or responsibilities. These choices result in a flat architecture, like toppings on a pizza.

kleine gekapselte Komponenten,  
gute Wartbarkeit,  
gute Wiederverwendbarkeit

monolytisch,  
hoch vernetzt  
schlecht wartbar

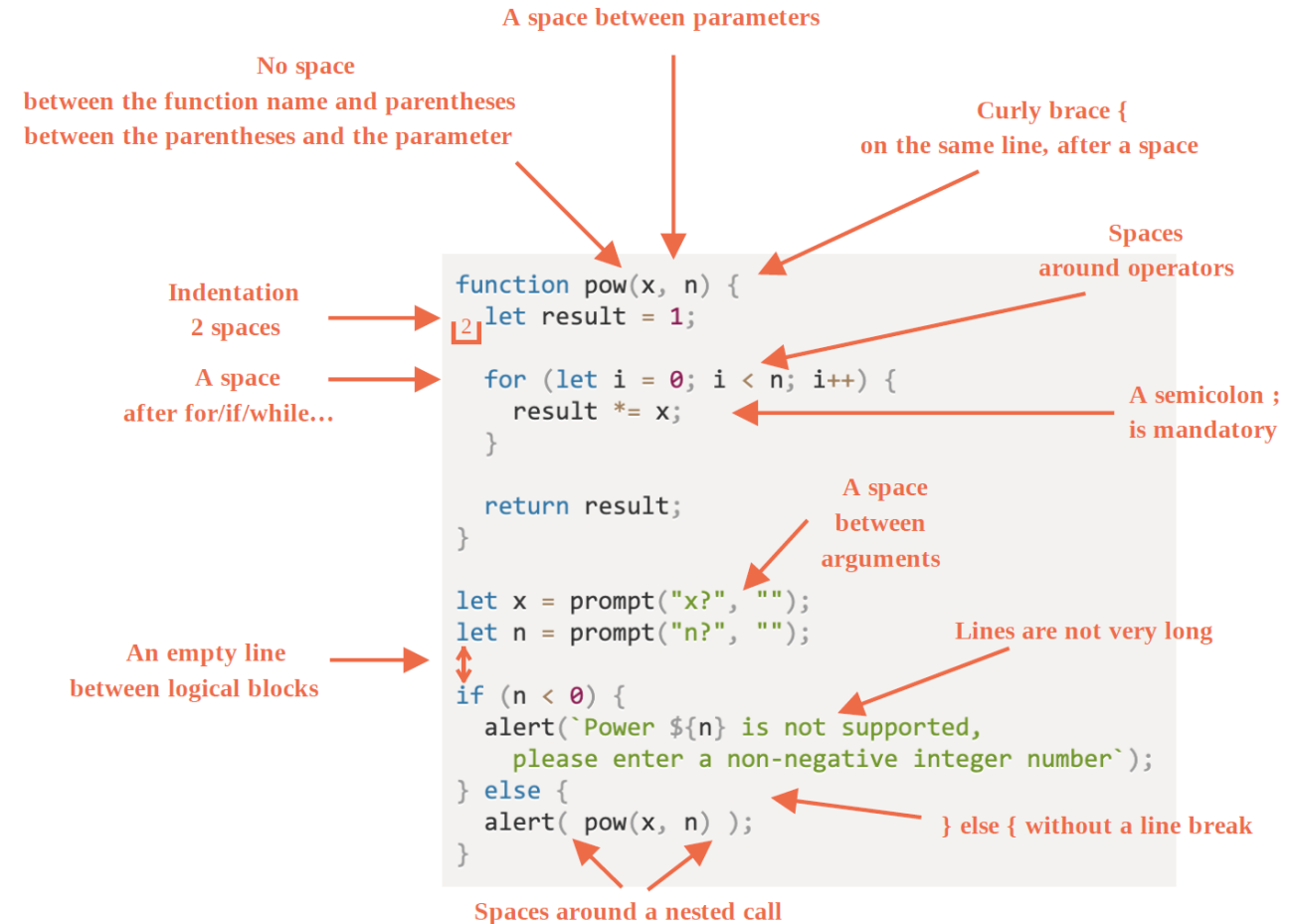
# Coding Practices

- ❑ Zu Beginn eines Entwicklungsprojektes sollten Sie einen sogenannten [Styleguide](#) für ihren [Source Code](#) erstellen.
- ❑ Der [Styleguide](#) enthält allgemeine Regeln zum Schreiben von Code
  - Wie muss der Source-Code kommentiert werden.
  - Bezeichnung von Variablen, Konstanten, Objekten, Funktionen, ...
  - Wieviele Leerzeichen zum Einrücken
  - Maximale Zeilenlänge (typisch 80 Zeichen)
  - usw.
- ❑ Ein Styleguide sorgt für ein [einheitliches Aussehen](#) des [Sourcecodes](#). Jedes Teammitglied kann dann den Source-Code anderer Teammitglieder leicht lesen.

- ❑ Im Internet stehen diverse Styleguides zur Verfügung.
- ❑ Motto: "Von den Großen lernen"  
[Googles JavaScript Styleguide](#)  
[JavaScript Standard Style](#)

# Coding Practices

- ❑ Öffnende { - Klammern in derselben Zeile mit Leerzeichen vom Identifier getrennt.
- ❑ Schließende } – Klammern in separater Zeile
- ❑ Lesbarkeit durch kurze Zeilenlängen (< 80 Zeichen).
- ❑ Einzüge (Indents)
  - Horizontal via Tab oder Leerzeichen ( 2 – 4 ) um Code innerhalb eines Code-Blocks { ... } zu kennzeichnen.
  - Vertikal via Leerzeile, um logische Code-Blöcke voneinander sichtbar zu trennen.
- ❑ "Helper"-Funktionen, die von einer Funktion verwendet werden, werden im Anschluss an den Quellcode der aufrufenden Funktion gelistet.



@Bild: <https://javascript.info/coding-style>

# Coding Practices

- ❑ **Funktionen** sind die wichtigsten „Bausteine“ eines Programmes. Sie ermöglichen den **mehrfachen Aufruf** eines Programm-Codes, **ohne** diesen zu **wiederholen** (DRY-Prinzip).
- ❑ Funktionen sind immer mit **Aktionen** verbunden. Daher ist ihr **Name** normalerweise ein **Verb**.
- ❑ Der **Name** sollte so kurz und so genau wie möglich sein und beschreiben, was die **Funktion tut**. Unterstützt die Lesbarkeit ihres Programmcodes.
- ❑ Best Practise: Funktionsnamen mit einem **verbalen Präfix** zu **beginnen**, das die **Aktion kategorisiert**. Die Präfixe werden vor Beginn eines Softwareprojektes in einem **Style-Guide** definiert.

## ❑ Beispiel für **Präfixe**:

- "get..." - get a value,
- "set..." - set a value,
- "calc..." - calculate something,
- "show..." - shows something,
- "check..." - check something,
- "create..." - create something

## ❑ Der **Wortstamm** konkretisiert die Aktion

```
showMessage(..)    // shows a message
getAge(..)         // retrieve the age
setPassword(..)    // set a password
calcSum(..)        // calculates a sum
createForm(..)     // creates a form
checkPermission(..) // checks a permission
```

# DRY-Principle

- ❑ DRY-Prinzip (**Don't repeat yourself**): Programmiercode sollte **nicht unnötigerweise** dupliziert werden.
  - **Grund:** Wenn eine Folge von Instruktionen mehrfach in ähnlicher Form verwendet werden, liegt diesen Anweisungen ein **allgemeineres Prinzip** zu, das explizit berücksichtigt und eingearbeitet werden sollte.
  - **Vorteile:**
    - ⇒ **Wartbarkeit:** Änderungen müssen nicht an mehreren, identischen Instruktionen vorgenommen werden.
    - ⇒ **Lesbarkeit:** Code ist leichter lesbar, da weniger Code existiert

# Coding Practices

- ❑ **Helper-Funktionen** werden verwendet
  - um bestimmte Aufgaben zu **verallgemeinern**, die in verschiedenen Teilen Ihrer Anwendung **wiederholt auftreten können**.
  - um den Programmcode durch Modularisierung **zu vereinfachen**.
- ❑ Best Practise ist es, **zuerst** den **Programmcode** zu schreiben, und dann die von ihm verwendeten "Helperfunktionen".
- ❑ **Hintergrund:** Wenn Programmcode gelesen wird, versuchen Sie zuerst ein **Gesamtverständnis** zu erlangen. Wenn der Code an erster Stelle steht, und für die Helper-Funktionen sprechende Programmnamen verwendet wurden, verlieren Sie sich beim initialen Lesen nicht in unnötige Details.

```
function showPrimes(n) {  
  nextPrime:  
  for (let i = 2; i < n; i++) {  
  
    // check if i is a prime number  
    for (let j = 2; j < i; j++) {  
      if (i % j == 0) continue nextPrime;  
    }  
  
    alert(i);  
  }  
}
```

**continue:** beendet aktuellen Schleifendurchlauf und wechselt zur nächsten Schleife

zerlegen in  
2 Funktionen

```
function showPrimes(n) {  
  
  for (let i = 2; i < n; i++) {  
    if (!isPrime(i)) continue;  
  
    alert(i);  
  }  
}  
  
function isPrime(n) {  
  for (let i = 2; i < n; i++) {  
    if (n % i == 0) return false;  
  }  
  
  return true;  
}
```



# Coding Practices

- ❑ **Beispiel:** Anstatt 3 einzelne Objekte für 3 Personen anzulegen und diese einzeln auf ihre ID zu prüfen:

```
const person1 = { };
const person2 = { };
const person3 = { };

main();

function main() {

    person1.firstName='Alice';
    person1.lastName='Spring';
    person1.id='0001';

    person2.firstName='Bob';
    person2.lastName='Sommer';
    person2.id='0002';

    person3.firstName='Carol';
    person3.lastName='Winter';
    person3.id='0003';

}
```

- ❑ Ist es besser eine **Personenklasse** und ein **Personen-array** zu verwenden, durch das man per for-loop iterieren kann

```
const personArray = [];

class Person {
    constructor(vName, nName, id) {
        this.firstName = vName;
        this.lastName = nName;
        this.id = id; }
};

main();

function main(numberPerson) {

    personArray[0] = new Person('Alice', 'Spring', '0001');
    personArray[1] = new Person('Bob', 'Sommer', '0002');
    personArray[2] = new Person('Carol', 'Winter', '0003');
}
```

# Coding Practices

- ❑ **Kommentare** im Quellcode sollten **kurz** sein und das **wesentliche** beschreiben.
- ❑ Sie können hierzu **JSDoc-Tags (siehe Beispiel)** verwenden, die zu einem einheitlichen, strukturierten und damit allgemein verständlichen Kommentarstil führen, der
  - menschen- und maschinenlesbar
  - über Tools eine Programmdokumentation auf Basis dieser Kommentare automatisch generierbar ist
  - von automatischen **Code-Analyser-Tools** verwendet werden kann, um den Code auf Fehler zu prüfen oder Tipps bei der Code-Erstellung anzubieten.
- ❑ JSDoc verwendet Tags, die mit einem **@** beginnen und einem **Schlüsselwort** zur Kennzeichnung der Bedeutung.

<https://jsdoc.app/>

```
/**
 *
 * Returns sum of 2 variables
 *
 * @author Luke Skywalker
 *
 * @function addition
 * Adds to numbers
 * @param {number} a First number to add.
 * @param {number} b Second number to add.
 * @returns {number} The Sum a + b.
 *
 */
function addition(a, b) {

    return a + b;

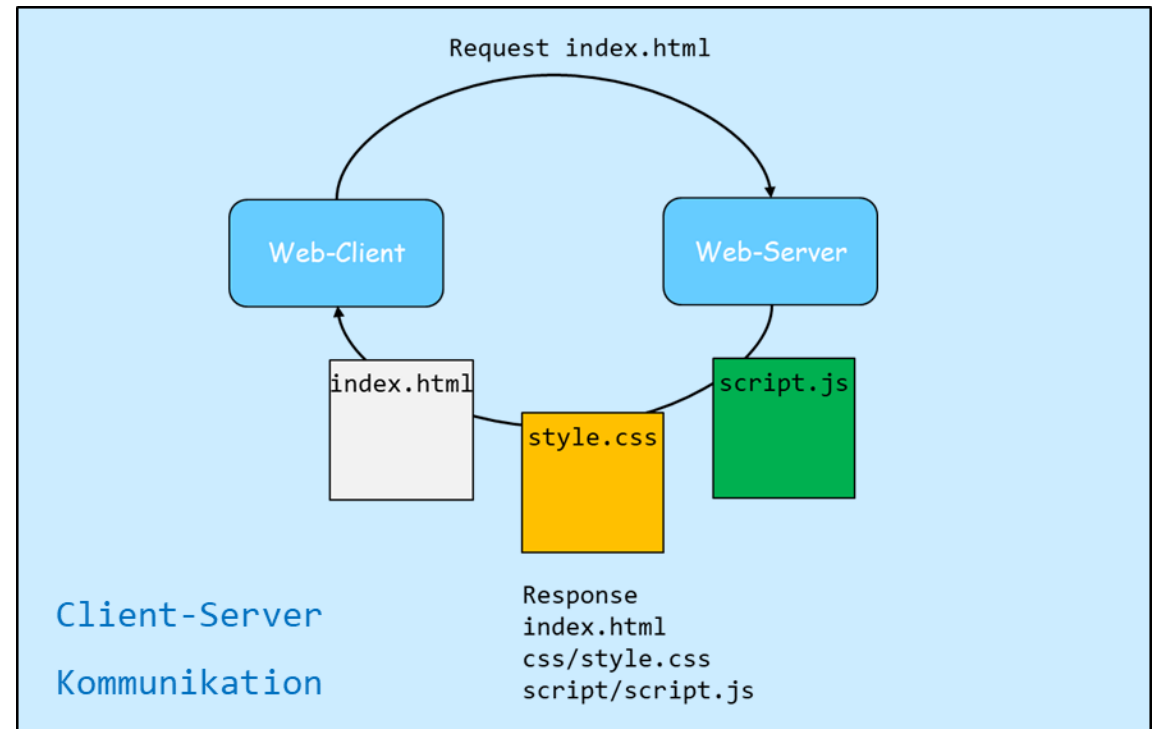
}
```

@author : Entwickler der Funktion

@param : Datentyp eines Parameters

@function : Name der Funktion

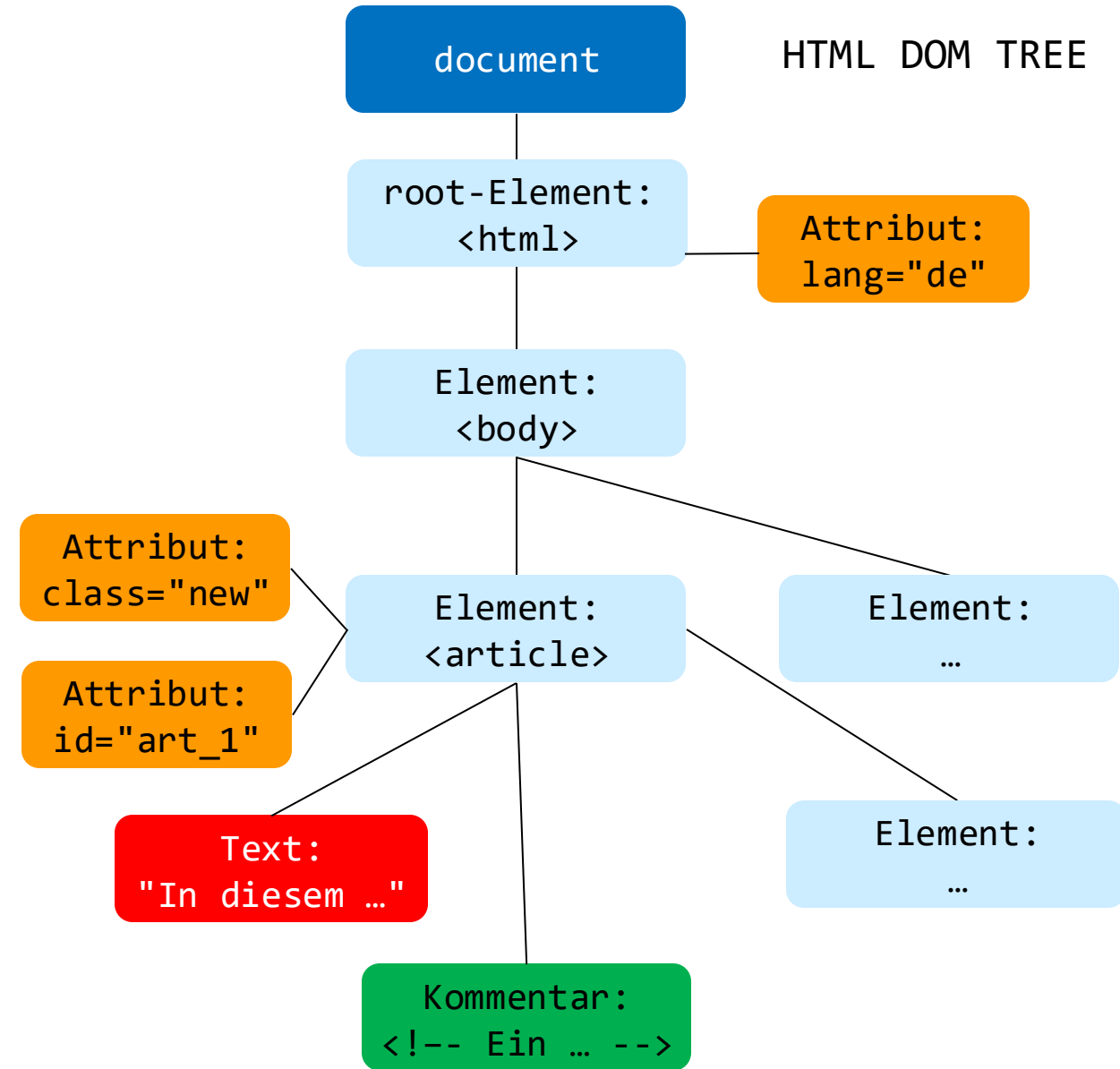
## 5.5 Document Object Model (DOM)



# Dynamische Webseiten mit JavaScript

- ❑ Wenn der Browser eine Webseite lädt, generiert er im Memory einen **HTML DOM Tree**.
- ❑ Die Knoten (**Nodes**) eines DOM-Trees bestehen aus
  - **Elementknoten** (Elemente)
  - **Attributknoten** (Attribute eines Elementes)
  - **Textknoten** (Textinhalt eines Elementes)
  - **Kommentarknoten** (Kommentar in einem Element)
- ❑ JavaScript kann über die **HTML DOM Schnittstelle** auf alle **HTML-Elemente** und deren **HTML-Attribute** zugreifen und diese **lesen, ändern, hinzufügen oder löschen**.
- ❑ Der Live DOM Tree einer Webseite kann mit dem folgenden Tool angezeigt werden:

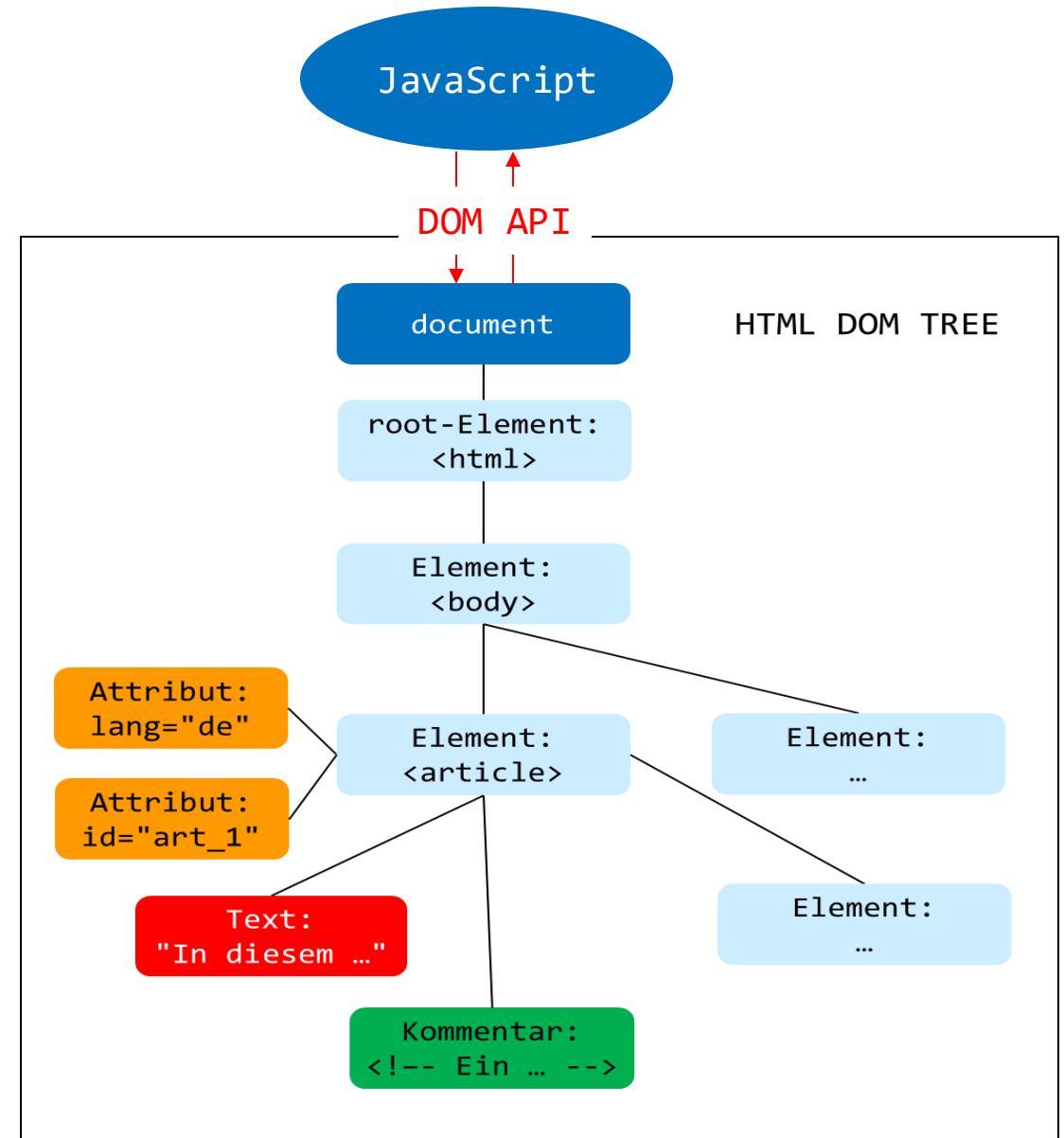
[Live DOM Viewer](#)



# DOM API

- Der Zugriff auf den DOM Tree erfolgt über die DOM API. Diese API besteht aus den Methoden des `document`-Objektes.
- Diese Methoden ermöglichen die Selektion einer Liste von Nodes (`NodeList`), oder einer Liste von HTML Elementen (`HTMLCollection`).
- Auf die Listenelemente kann über eckige Klammern und `Index` oder über eine `for-of`-Schleife analog zu einem `JavaScript-Array` zugegriffen werden.

```
const element =  
    document.getElementById('elementid');  
const element =  
    document.getElementsByClassName('classname')[0]
```



# NodeList versus HTMLCollection

- Sowohl NodeList als auch HTMLCollection sind Array-ähnliche Objekte in JavaScript, die Sammlungen von DOM-Knoten darstellen. Es gibt jedoch einige Unterschiede zwischen ihnen:

- **NodeList:**

- NodeList ist eine Sammlung von Knoten (**Nodes**), die von verschiedenen DOM-Methoden wie beispielsweise **childNodes** zurückgegeben werden.
- NodeList ist eine Live-Darstellung der HTML-Seite.
- NodeList-Objekte können jede Art von Knoten enthalten: **Elementknoten**, **Textknoten**, **Kommentarknoten**, ... usw.
- Dient zur **Evaluierung** und **Steuerung** der Baumstruktur.

- **HTMLCollection:**

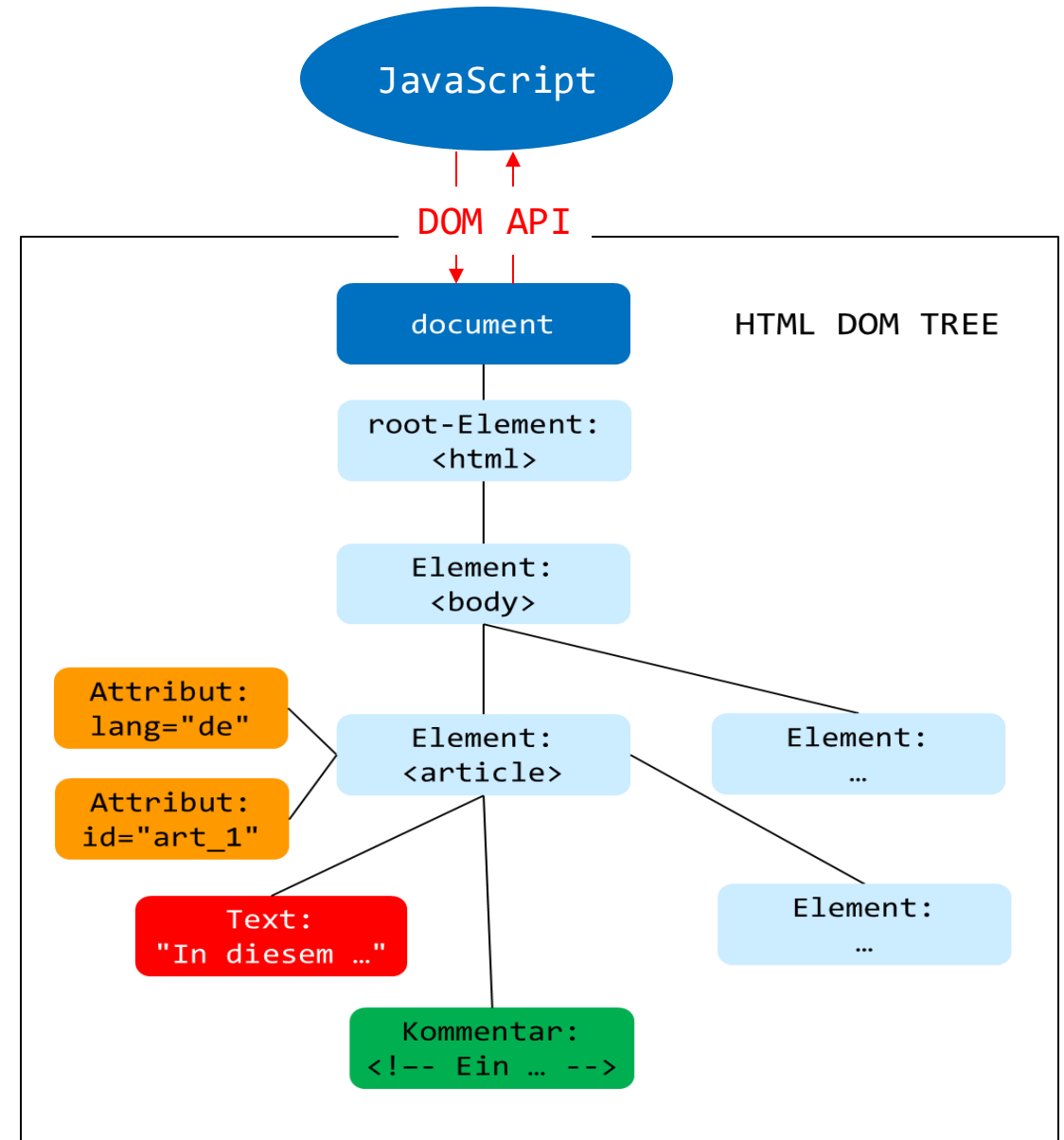
- Bei HTMLCollection handelt es sich um eine Sammlung von HTML-Elementen (sog. **Element Nodes**), die von Methoden wie **getElementById** oder **getElementsByClassName** zurückgegeben werden.
- HTMLCollection enthält ebenfalls eine Live-Darstellung der HTML-Seite.
- HTMLCollection-Objekte enthalten nur Elementknoten.
- Dient zur Veränderung des Inhaltes und des Designs HTML-Elementen und HTML-Attributen.

# DOM API

- Die per DOM erzeugten Elemente stellen **JavaScript-Objekte**, mit **Eigenschaften** und **Methoden**:

```
//Textinhalt eines Elementes
//Eigenschaft innerText
element.innerText="Neuer Text xyz"
//Eigenschaft innerHTML
element.innerHTML="Neuer <em>Text</em> xyz"
//Methode appendchild('p')
//(Child <p> an element anhängen
element.appendChild('p');
```

- `.innerText="..."`: nur einfacher Text erlaubt
- `.innerHTML="..."`: einfacher Text und inline-Elemente zur Formatierung erlaubt (e.g. `<em>`)



# Im DOM-Tree navigieren

- ❑ Sie können im DOM-Tree ausgehend von einem beliebigen Node oder Element nach oben bzw. nach unten navigieren.
- ❑ Der oberste Einstiegspunkt in den DOM-Tree ist durch das Objekt `document.documentElement` gegeben, das die Webseite (`<html>`) darstellt.
- ❑ Ausgehend von einzelnen Elementen kann Sie nach unten, mittels

```
element.childNodes;           //Liste aller Kind-Knoten
```

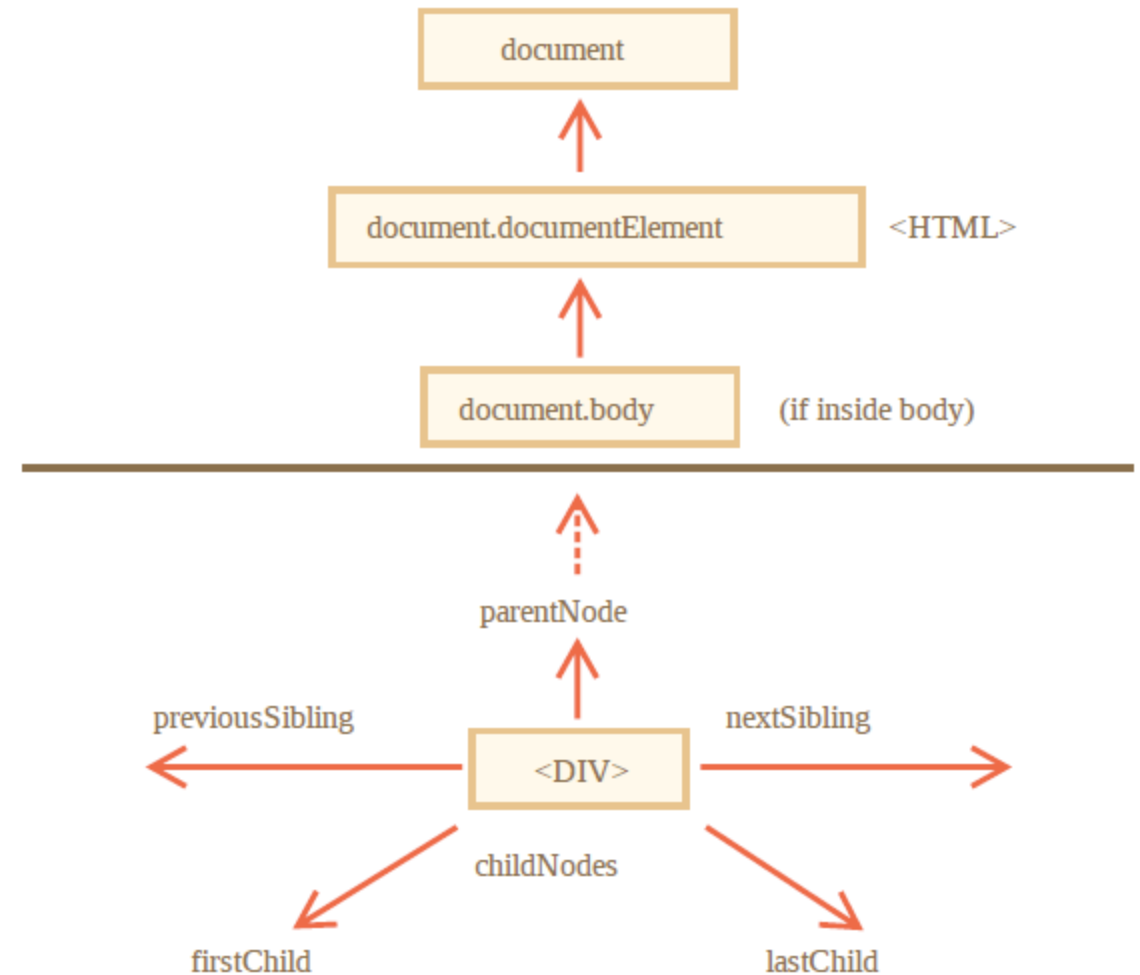
navigieren, bzw. mit

```
element.parentNode;           //Eltern-Knoten
```

nach oben wandern.
- ❑ Geschwisterelemente können mittels

```
element.nextSibling;           //nachfolgendes Gesch
```

```
element.previousSibling;       //vorangehendes Element
```





# DOM: Direct Access

- ❑ Auf bestimmte Elemente des DOM-Tree kann **direkt** **zugegriffen** werden.
- ❑ **Beispiel:** Das root-Element `<html>` einer Webseite erhält man direkt über `document.documentElement`

```
const rootElement = document.documentElement;
const firstTier = rootElement.childNodes;
// firstTier: direct childs of root
for (let i = 0; i < firstTier.length; i++)
{
    console.log(firstTier[i].nodeName);
    //element name: HEAD, BODY, #text
    console.log(firstTier[i].textContent);
    //Textinhalte der childs
}
```

- ❑ Weitere **direkte** Zugriffsmöglichkeiten sind beispielsweise :  

<code>document.documentElement</code>	: <code>&lt;html&gt;</code> -Element
<code>document.body</code>	: <code>&lt;body&gt;</code> -Element
<code>document.head</code>	: <code>&lt;head&gt;</code> -Element
<code>document.title</code>	: <code>&lt;title&gt;</code> -element
<code>document.URL</code>	: URL der Seite
<code>document.cookie</code>	: Cookie der Seite
<code>document.lastModified</code>	: letzte Seitenänderung
<code>document.inputEncoding</code>	: Seiten Encoding
<code>document.forms</code>	: alle <code>&lt;form&gt;</code> -Elemente
<code>document.images</code>	: alle <code>&lt;img&gt;</code> Elemente
<code>document.links</code>	: alle <code>&lt;a&gt;</code> - Elemente
<code>document.scripts</code>	: alle <code>&lt;script&gt;</code> - Elemente
...	

# DOM Queries: ID-Selektor

- ❑ Es gibt verschiedene sogenannte DOM Queries, um die Elemente in einem DOM-Tree zu selektieren.
- ❑ Will man genau ein Element herausgreifen, verwendet man den **ID-Selektor**.

```
//Element mit id="mytest" als Objekt erzeugt
const obj = document.getElementById("mytest");
//Ändert den Inhalt des Elements durch hinzufügen
// von HTML-Code
obj.innerHTML = "<em>Innerer Text</em> ist gleich";
//Setzt die Schriftfarbe auf blau
obj.style.color = "blue";
```

# DOM-Queries: Tag-Selektor und Navigation

- ❑ Tag-Selektor liefert eine `HTMLCollection` aller Elemente desselben `HTML-Tags`.
- ❑ Beispiel: Alle `<article>`-Elemente innerhalb eines `HTML-Elementes` erhält man mittels

```
const artList =  
    document.getElementsByTagName('article');
```

- ❑ Beispiel: Ausgehend vom `ersten <article>-Element` auf ihrer Webseite, bestimmen Sie sein `unmittelbares Geschwisterelement` `nextSibling`. Handelt es sich bei diesem um einen `Artikel`, weisen Sie diesem Element das `class-Attribut "article--content"` zu ansonsten `class-Attribut "other--content"`.

```
//first article element  
const myArticle =  
    document.getElementsByTagName('article')[0];  
//next sibling  
const firstSibling = myArticle.nextSibling;  
if (firstSibling.nodeName === 'article'){  
    //if sibling is of type article  
    firstSibling.className="article--content"  
} else {  
    firstSibling.className="other--content"  
}  
  
//select next sibling from firstSibling  
const nextSibling = firstSibling.nextSibling;  
nextSibling.className="text--content";
```

# DOM Queries: Class-Selektor

- ❑ **Class-Selektor:** Liefert eine **HTML-Collection** aller Elemente einer **bestimmten HTML-Klasse**.
- ❑ Über die Methode **element.item(Index)** oder über **element[Index]** können die Elemente an einer bestimmten Array-Position **Index** selektiert werden.
- ❑ Mittels **element.nodeName** wird der Name des Elementes ausgegeben.
  - Im Beispiel: ARTICLE , P, EM
- ❑ **Beispiel:** Elemente der Klasse "important" wird ein neuer Textinhalt zugewiesen:

Beispiel: HTML-Auszug

```
<article class="important"></article>
<p class="important" name="p1"></p>
<em class="important" id="e1"></em>
```

```
const elements =
    document.getElementsByClassName('important');

for(let element of elements) {
    console.log(element.nodeName);
}

if (elements.length > 1) {
    const element0 = elements[0]; // <article>
    element0.innerText = "Ich bin article";
    const element1 = elements[1]; // <p>
    element1.innerText = "Ich bin p";
    const element2 = elements.item(2); // <p>
    element2.innerText = "Ich bin em";
}
```

# DOM Queries: CSS-Selektor

- ❑ **CSS-Selektor:** Liefert alle **Knoten** in Form einer **NodeList**.
- ❑ **querySelector()** gibt das erste Element im Dokument zurück, das mit dem angegebenen CSS-Selektor (**p.c1**) übereinstimmt. Kein Treffer → null zurückgegeben.

Beispiel: HTML-Auszug

```
<section class="c1"> </section>
<article class="c1"> </article>
```

```
const elem = document.querySelector('section.c1');
console.log(elem.nodeName); // section
```

- ❑ **querySelectorAll()** gibt eine statische NodeList zurück, das mit dem angegebenen CSS-Selektor übereinstimmen.
- ❑ Die Methode **forEach()** ruft eine Funktion für jedes Element **item** in einem Array auf.
- ❑ Die Methode **forEach()** wird für leere Elemente nicht ausgeführt.

Beispiel: <section class="c1">, <article class="c1">

```
const nodes = document.querySelectorAll('.c1');
nodes.forEach(item => console.log(item.nodeName));
//section article
```

# Inhalte und Attribute ändern

## □ Inhalte von Elementen ändern

```
element.innerText = "Ein neuer Text.";
element.innerHTML
    ="Ein neuer <em>Text</em> mit HTML-Content.";
```

## □ ID eines Elementes ändern, ID entfernen

```
element.id = 'neue id';
element.removeAttribute('id');
```

## □ Attribute abfragen und setzen

```
//Getting an attribute
let lang = element.getAttribute('lang');
//Setting Attribute "mystatus"
element.setAttribute('mystatus', 'ok');
```

```
element.className = 'new';
```

## □ Attribute eines Elementes prüfen und entfernen

```
//Check if attribute "mystatus" existiert
(true, false)
if (element.hasAttribute('mystatus') {
    //Remove attribute
    element.removeAttribute('mystatus');
}
```

"Good practise"

# Gestalt ändern

- ❑ In vielen Fällen empfiehlt es sich, über das **Klassen-Attribut** **dynamisch** das **Erscheinungsbild** anzupassen.
- ❑ Dazu legen Sie vorab für **verschiedene Klassen** (class-Attribute) ein **Erscheinungsbild** in ihrem CSS-Stylesheet fest.
- ❑ Im Verlauf der Programmlogik verändern Sie dann dynamisch den Namen des **class**-Attributes.
- ❑ **Vorteil:** "Separation of Concerns" Architekturprinzip wird beibehalten.

- ❑ Um die **CSS-Gestaltung** eines Elementes zu ändern können Sie auch sein inline style-Attribut verwenden:

```
//Setzen der Schriftgröße auf 20pt  
element.setAttribute('style', 'fontsize:20pt;');  
element.style.fontSize = "20pt";
```

# Gestalt ändern mit class- oder style-Attribut

- ❑ In vielen Fällen empfiehlt es sich, über **Klassen dynamisch** das **Erscheinungsbild** anzupassen.
- ❑ Dazu legen Sie vorab für **verschiedene Klassen** (class-Attribute) ein Erscheinungsbild in ihrem CSS-Stylesheet fest.
- ❑ Im Verlauf der Programmlogik verändern Sie dann dynamisch den Namen des **class**-Attributes.
- ❑ **Vorteil:** "Separation of Concerns" Architekturprinzip wird beibehalten.

- ❑ Um die **CSS-Gestaltung** eines Elementes zu ändern können Sie auch sein inline style-Attribut verwenden:

```
//Setzen der Schriftgröße auf 20pt  
element.setAttribute('style', 'fontsize:20pt;');  
element.style.fontSize = "20pt";
```



# JavaScript und CSSOM-Tree

- ❑ Um den **CSSOM-Baum** im Memory des Browsers zu verändern, werden zuerst alle **style**-Objekte per DOM-Schnittstelle bestimmt.
- ❑ **style**-Objekte sind
  1. per <link> eingebundene CSS-StyleSheets
  2. per <style>-Tag in HTML definierte CSS-Regeln
- ❑ **Beispiel:** Hintergrundfarbe in der ersten Regel des ersten StyleSheets ändern

```
//Erstes Style-Objekt == CSS-Datei
const stylesheet = document.styleSheets[0];
//Verändern der Hintergrundfarbe
stylesheet.cssRules[0].style.backgroundColor =
"blue";
```

- ❑ **Beispiel:** Alle Regeln in eine **CSSRule**-Liste einlesen und nach bestimmter Regel in allen StyleSheets suchen und anschließend ein Attribut setzen oder verändern.

```
//Alle Regeln aus erstem Style-Objekt
const myRules =
    document.styleSheets[0].cssRules;

//Suche nach Regel mit Selektor "#btn"
for (const myRule of myRules) {
    if (myRule.selectorText == '#btn') {
        myRule.style.color = 'red';
    }
}
```

# JavaScript und CSS-Variablen

- ❑ CSS –Variablen können direkt mit dem style-Attribut dynamisch überschrieben werden.

## CSS-Datei

```
:root {  
  --main-bg-color: lightblue;  
  --main-text-color: black;  
}
```

## JS-Datei

```
document.documentElement.style.setProperty(  
  "--main-bg-color ", "blue");
```

# HTML-Elemente: Hinzufügen, Ändern und Entfernen

- ❑ Mittels JavaScript und DOM lassen sich dynamisch HTML-Elemente erzeugen, entfernen, ersetzen, hinzufügen ... (siehe Tabelle).
- ❑ **Beispiel:** Hinzufügen eines Buttons zu einem HTML-Element

```
//Erzeugen eines Elementes (hier button)
btn = document.createElement('button');
//Beschriften des Buttons
btn.innerHTML = "CLICK ME";

//Selektion eines Elementes (Formular)
objForm = document.getElementById("myform");
//Hinzufügen des Buttons zum selekt. Element
objForm.appendChild(btn);
```

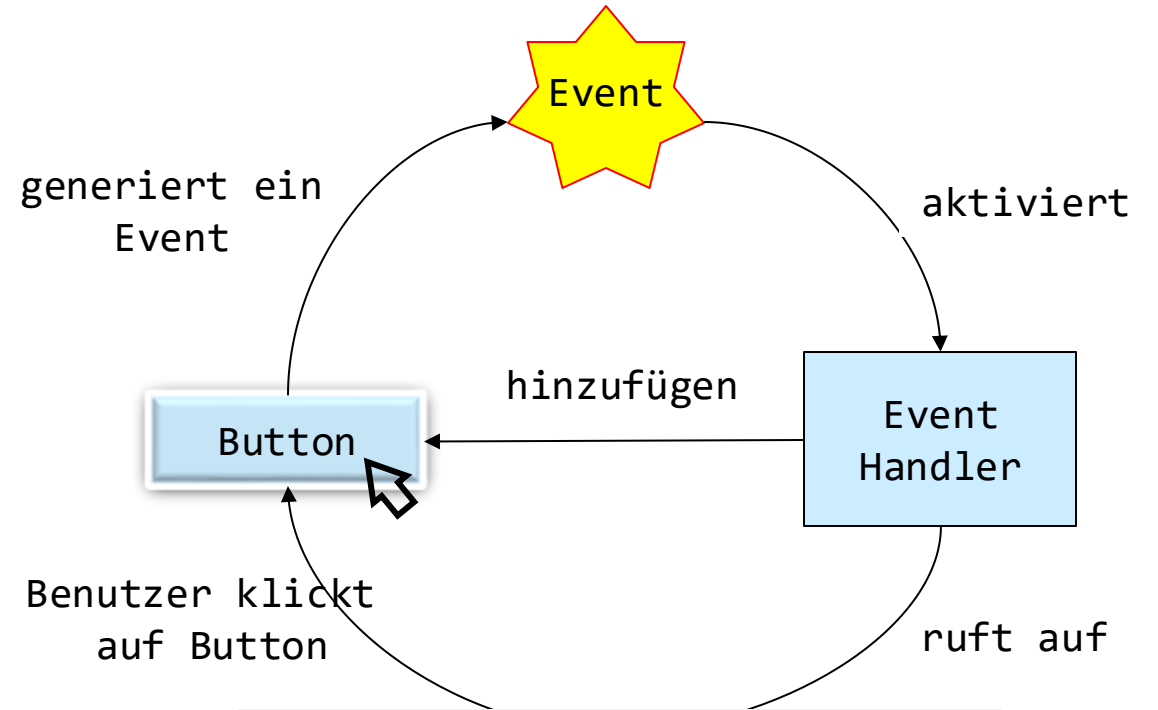
- ❑ **Beispiel:** Entfernen eines HTML-Elementes

```
//Suche des zu entfernenden Elementes (id)
const elemObj = document.getElementById('myid');
//Selektion Elternelement
const parent = elemObj.parentNode;
//Entfernen des Elementes
parent.removeChild(elemObj);
```

Method	Description
document.createElement( <i>type</i> )	Erzeugt ein HTML Element
document.removeChild( <i>element</i> )	Entfernt ein HTML Element
document.appendChild( <i>element</i> )	Ergänzt ein HTML Element
document.replaceChild( <i>new</i> , <i>old</i> )	Ersetzt ein HTML Element
document.insertBefore ( <i>new</i> , <i>exist</i> );	HTML Element vor anderes

# Events

- Events sind Ereignisse, wie z.B.
  - das Anklicken eines Buttons,
  - die Tastatureingabe in ein Textfeld,
  - das vollständige Laden eines Bildes,
  - ...
- Jedes HTML-Element kann Events auslösen.
- Damit ein Element auf ein Ereignis reagieren kann, muss für das jeweilige Event ein Event-Handler erstellt werden, der beim Eintreten eines Events eine JavaScript-Callback Funktion ausführt.
- Event-Handler können entweder
  - direkt im HTML-Code,
  - oder im JavaScript-Codeangelegt werden.

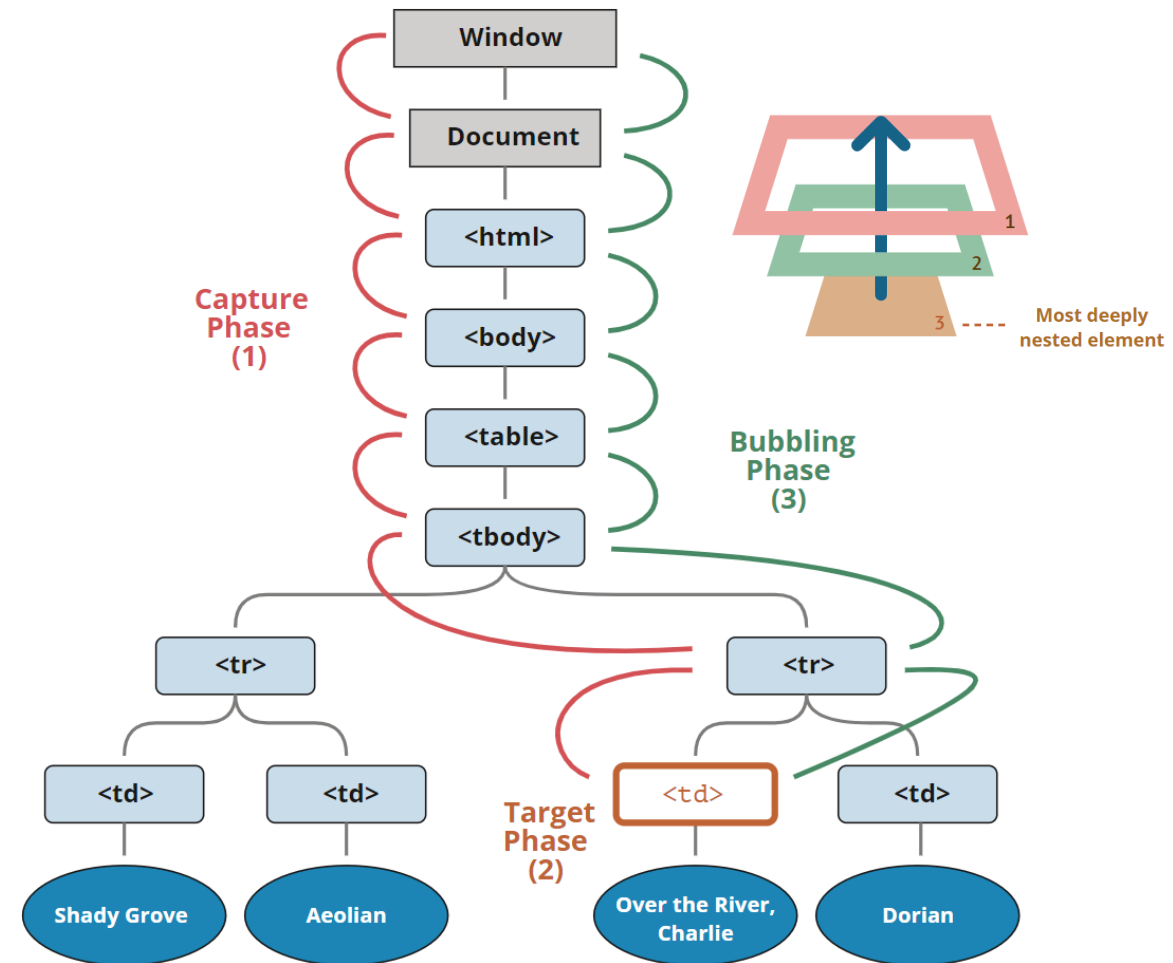


```
<script>
btn=document.getElementById('btnhoverme');
btn.addEventListener('mouseover',
    function ()
    {btn1.style.backgroundColor= "blue";}
);
</script>
```

JavaScript ändert den Inhalt  
der Webseite

# Event Propagation

- ❑ Die Standard-DOM-Ereignisse beschreiben 3 Phasen der Ereignisweitergabe:
  - **Erfassungsphase (Capturing)**: Das **Event** wird über das Fenster, zum HTML-Dokument, über den DOM-Tree bis zum einzelnen betroffenen Element weitergereicht.
  - **Zielphase (Target)**: Das Ereignis hat das Zielement erreicht.
  - **Sprudelphase (Bubbling)**: Das Ereignis sprudelt aus dem Element über den DOM Tree nach oben.
- ❑ Event-Listener können sowohl für die **Bubbling-Phase** (`value: false`) oder für die **Capturing-Phase** (`value: true`) registriert werden.



# Event Handler im HTML-Code

- ❑ **Erste Methode:** Im HTML-Element wird der Event-Handler direkt codiert (**Bad Practise**).
- ❑ Event wird in der Bubbling-Phase erfasst.
- ❑ Beispiel: HTML-Button mit Event-Handler **onclick**, der bei einem Mausklick ein Info-Fenster anzeigt:

```
<button onclick="alert('Action!')">Senden</button>
```

- ❑ Beispiel: HTML-Button mit Event-Handler **onmouseover**, der Button via **JavaScript** rot einfärbt

```
<input id='btnhoverme' value="Senden" type="button" onmouseover="hoverMe(this)" >
```

- ❑ Eine Liste der **HTML-Events** finden Sie beispielsweise auf der W3C-Schools Seite:

[https://www.w3schools.com/tags/ref\\_eventattributes.asp](https://www.w3schools.com/tags/ref_eventattributes.asp)

```
<script>
    function hoverMe(object){
        object.style.backgroundColor= "red";
    }
</script>
```

# Event Handler im JavaScript-Code

- ❑ **Zweite Methode:** Der Event-Handler wird im JavaScript-Code definiert (**Good Practise**).
- ❑ In HTML wird kein Eventhandler benötigt.
- ❑ Stattdessen wird mit der Methode `addEventListener` ein Event-Handler für das `mouseover` – Event in **JavaScript** generiert.
- ❑ Beispiel: Argumente von `addEventListener`
  - DOM-Event: `mouseover`
  - Callback-Funktion: `hoverMe`
  - Event Flow (optional):  
`false // == default → bubbling phase`
- ❑ Eine Liste der **HTML DOM Events** finden Sie auf der W3C-Scholls Seite  
[https://www.w3schools.com/jsref/dom\\_obj\\_event.asp](https://www.w3schools.com/jsref/dom_obj_event.asp)

```
<input id='btnhoverme' value="Hover Me"
      type="button">
```

```
<script>
  btn=document.getElementById('btnhoverme');
  btn.addEventListener('mouseover',
    hoverMe,
    false);
  function hoverMe(){
    btn.style.backgroundColor= "blue"
  }
</script>
```

CallBack-Funktion

# Event Handler und asynchrone Code-Ausführung

- ❑ Die Funktion im Event-Handler wird ohne die **Klammern ()** angegeben. Dies signalisiert dem Interpreter, dass es sich um eine **Callback-Funktion** handelt.
- ❑ **Andere Möglichkeit:** Verwendung von **anonymen Funktionen**. Dies ermöglicht eine Funktion mit **Parametern** aufzurufen:

```
<script>
  btn=document.getElementById('btnhoverme');
  btn.addEventListener(
    'mouseover',
    function () {hoverMe('blue');},
    false
  );
</script>
```

```
function hoverMe(myColor) {
  btn.style.backgroundColor= myColor;
}
```



# Event Handler und asynchrone Code-Ausführung

- Besitzt die Funktion nur einen geringen Code-Umfang, können Sie auch **nur** mit der **anonymen Funktion** arbeiten, ohne zusätzlich eine Funktion zu definieren.

```
<script>
btn=document.getElementById('btnhoverme');
btn.addEventListener(
    'mouseover',
    function () {
        btn.style.backgroundColor= 'blue';},
    false );
</script>
```

- Noch kürzer geht es mit der **Arrow-Schreibweise**..

```
<script>
btn=document.getElementById('btnhoverme');
btn.addEventListener(
    'mouseover',
    () => {btn.style.backgroundColor='blue';},
    false );
</script>
```

# Formulare <form> und JavaScript

- Die Formulare einer Web-Seite werden verwendet, um mit einem Benutzer Daten auszutauschen.
- Alle **Formulare** einer Webseite können über eine HTML-Collection `document.forms` selektiert werden. Sie sind dort in der gleichen Reihenfolge wie im HTML-Dokument gespeichert.

// Erstes Formular einer Webseite

```
const objForm = document.forms[0];
```

//Formular mit dem Attribut **name="loginForm"**

```
const objForm = document.forms.loginForm;
```

//Element eines Formulars mit dem Attribut

**//name="passWd"**

```
const objElem = objForm.elements.passWd;
```

//Wert bzw. Inhalt eines Elementes lesen/ändern

```
let password= objElem.value;
```

```
objElem.value = 'Neuer Wert';
```

- Beispiel: Extraktion der Eingabewerte eines Login-Formulars

Felder mit \* sind erforderlich

<form name="loginForm">

Benutzername \*

name="userName"

Passwort \*

name="passWd"

Login

name="loginButton"

```
const objForm = document.forms.loginForm;  
const objButton = objForm.elements.loginButton;  
objButton.addEventListener(  
  'click',  
  function () {  
    let objUserName = objForm.elements.userName;  
    console.log(objUserName.value);  
  },  
  false);
```

# this - Keyword

- ❑ In objektorientierten Sprachen möchte man oft auf die **Eigenschaften** oder **Methoden** des **aktuellen Objektes** zugreifen. Dafür wurde das Schlüsselwort **this** geschaffen.
- ❑ In JS enthält **this** eine **Referenz** auf das **Objekt**, welches die aktuelle **Funktion** aufgerufen hat.
- ❑ Standardmäßig hat **this** in JS den Wert **undefined**.
- ❑ Im **Beispiel** auf der **rechten Seite** verweist **this** auf den **Login-Button**, der das Click-Ereignis ausgelöst hat und die Funktion `login ()` aufruft:

```
<form name="loginForm" ...>
  ...
  <button id="idloginbutton" value="Login"
    name="loginButton" type="button">
  </button>
</form>
```

- ❑ Über **this** können dann beispielsweise die Eigen-schaften des Buttons ausgegeben werden:

```
<script>
  let objForm = document.forms.loginForm;
  let objButton = objForm.elements.loginButton;
  objButton.addEventListener(
    'click',
    function () { alert('Tag: ' + this.tagName +
      ' ID: ' + this.id + ' Name: '
      + this.name + ' Value: ' +
      this.value);},
    false);
</script>
```

Auf localhost:63342 wird Folgendes angezeigt:

Tag: BUTTON ID: idloginbutton Name: loginButton Value: Login

Ok

# Event Object

- ❑ Wenn ein Ereignis eintritt, erstellt der Browser ein **Ereignis-objekt** (engl. **Event Object**) und übergibt dieses an den jeweiligen Event-Handler.
- ❑ Das Ereignisobjekt **event** enthält Details über das Event:
  - **event.type**: HTML DOM Event  
`click`, `keypress`, `mouseover`,...
  - **event.target**: das das Event auslösende Element
  - **event.property**: vom auslösenden Element abhängige Eigenschaften, beispielsweise
    1. **event.clientX**, **event.clientY**: Position der Maus in x- und y-Richtung
    2. **event.key**: gedrückte Keyboard-Taste
    3. **event.buttons**: gedrückte Maustaste

- ❑ Beispiel: Ausgabe der gedrückten Taste

```
<script>
  //Event für HTML-Dokument
  document.addEventListener(
    'keydown',
    (event) => {
      let name = event.key;
      // Alert the pressed key name on keydown
      alert('You have pressed key ' + name),
      false);
    }
  </script>
```

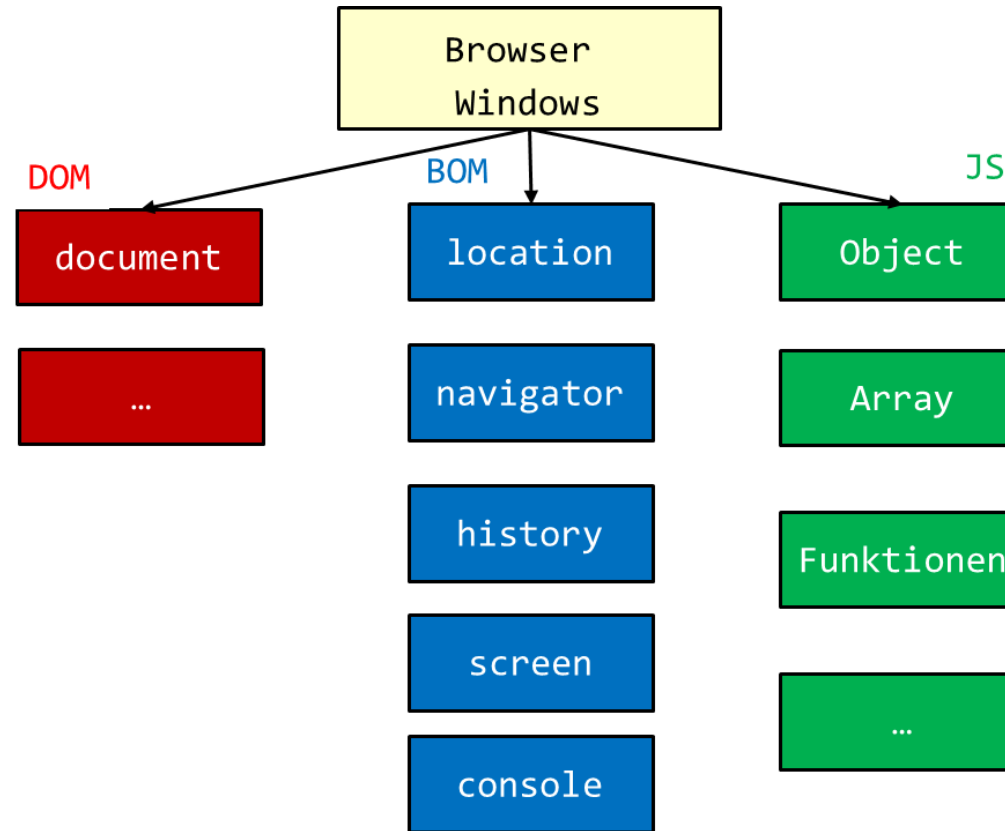
# Event Object

- Beispiel: Reaktion auf ein Maus-Event

```
document.addEventListener(  
  'mousedown',  
  (event) => {  
    let name = event.buttons;  
    // Alert the mouse button on mousedown  
    switch(name){  
      case 1:  
        alert('Pressed left button');  
        break;  
      case 2:  
        alert('Pressed right button');  
        break;  
      case 3:  
        alert('Pressed right & left button');  
        break;  
      case 4:  
        alert('Pressed middle button');  
        break;  
    } },  
  false);
```

Werden 2 Maustasten gleichzeitig gedrückt, wird von `event.buttons` der Summenwert ausgegeben.

## 5.6 Browser Object Model (BOM)

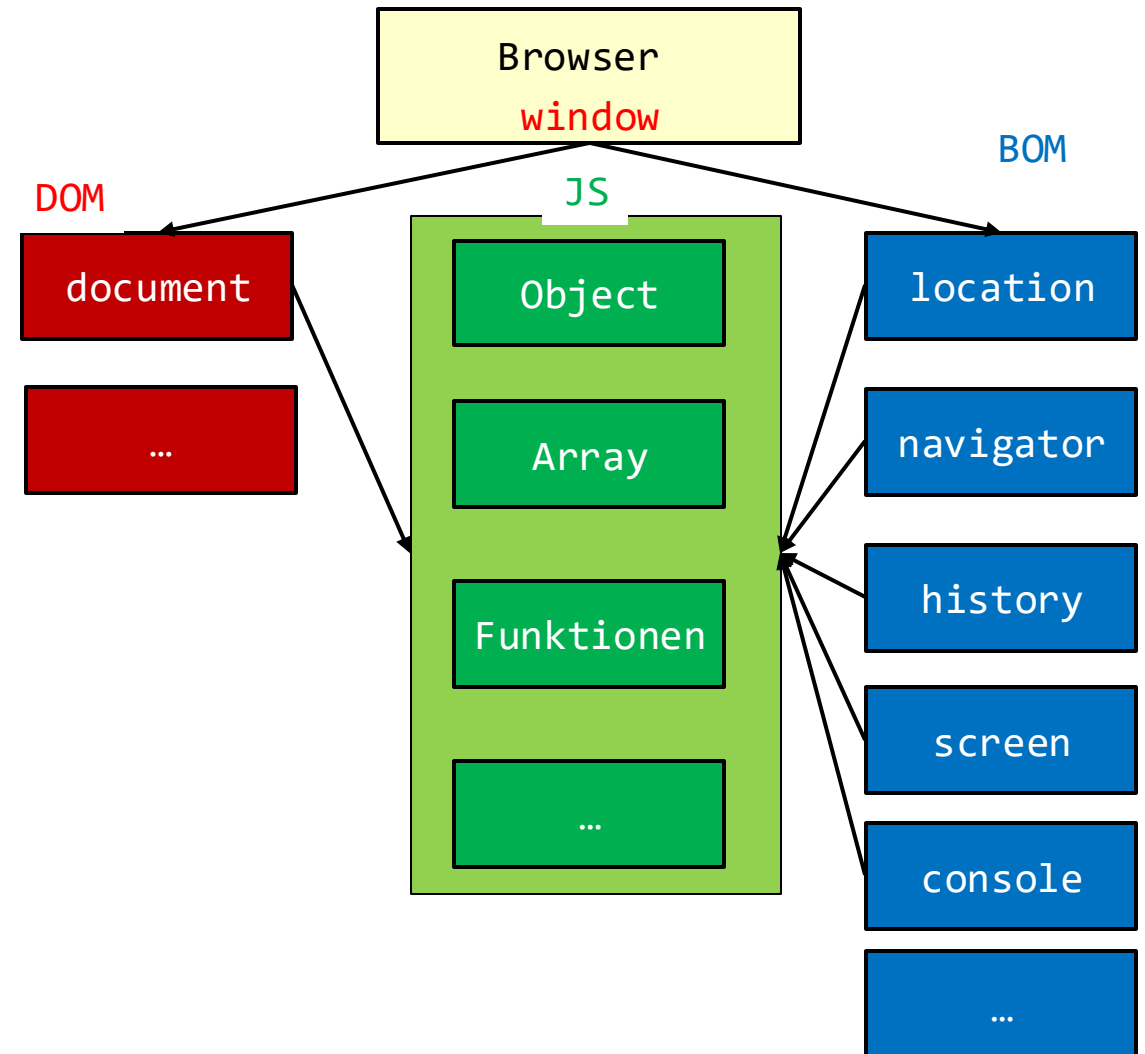


# Browser Object Modell (BOM)

- Ein Browser kann neben DOM der JavaScript-Engine weitere Funktionalitäten bereitstellen.
- Diese zusätzlichen Funktionalitäten werden unter dem Begriff **Browser Object Model (kurz BOM)** zusammengefasst.
- Die **DOM-Objekte**, die **BOM-Objekte** und die **JavaScript-Objekte**, sind Teil eines **root-Objektes** namens **window**.

```
window.document.getElementById("header");  
//equal to  
document.getElementById("header");
```

- Das **window-Objekt** repräsentiert dabei das **Browser-fenster** (Tab in einem Browser).
- Über das **window** und die BOM-Objekte kann per JS auf die **Laufzeit-Umgebung** der Webseite zugegriffen werden.



# window-Objekt

- ❑ Über das **window-Objekt** können Sie zum Beispiel die Höhe und Breite des Browser-Fenster (**Viewport**) ohne Scrollbar und Toolbars bestimmen

```
let w = window.innerWidth;  
let h = window.innerHeight;
```

- ❑ Eine nützliche Methode, die schon mehrfach gezeigt wurde, ist die Anzeige eines **modalen Dialogs** für **kritische Benachrichtungen** im Browser-Fenster

```
window.alert('Hallo, wie geht's?');  
//equal to  
alert('Hallo, wie geht's?');
```

- ❑ Die Methode **window.setTimeout()** stellt einen **TimeOut** für eine **Callback-Funktion** bereit.
- ❑ Beispiel: Öffnen eines **neuen Browser-Tabs** mit der Methode **window.open()** nach **3s** (3000ms)

```
window.setTimeout(  
    () => window.open('https://google.com'),  
    3000);  
  
//equal to  
setTimeout(  
    () => open('https://google.com'),  
    3000);
```



# location- und console- Object

- ❑ Das `location-Object` enthält Informationen zur aktuellen URL der Webseite.
- ❑ Beispielsweise:

```
//Ausgabe der Root-Url (ohne document path)
alert(location.origin);

//Ausgabe der aktuellen URL (all inclusive)
alert(location.href);

//Öffnet URL (in vorhandene Webseite)
location.href="https://www.w3schools.com";

// Öffnet neue Webseite (ersetzt vorhandene)
location.replace("https://www.w3schools.com");
```

- ❑ Weitere Properties und Methoden finden Sie unter [https://www.w3schools.com/jsref/obj\\_location.asp](https://www.w3schools.com/jsref/obj_location.asp)

- ❑ Für Debugging haben Sie das `console-Objekt` schon verwendet.
- ❑ Das `console-Object` besitzt eine Vielzahl an Methoden, um eine Webseite zu debuggen
- ❑ Beispielsweise:

```
//Ausgabe eines Textes auf die Console
console.log('Ein Test!');

//Ausgabe eines Fehlers auf die Console
console.error('Ein Fehler!');

//Ausgabe einer Warnung auf die Console
console.warn('Lange Ladezeiten');
```

- ❑ Weitere Methoden finden Sie unter [https://www.w3schools.com/jsref/obj\\_console.asp](https://www.w3schools.com/jsref/obj_console.asp)

# navigator-Object

- ❑ Das `navigator-Object` bietet Informationen zum `Status` und zur `Identität` des User Agent, der die aktuelle Webseite ausführt
- ❑ Die Property `userAgent` gibt den vom Browser an den Server gesendeten `User-Agent-Header` zurück. Der zurückgegebene Wert enthält Informationen über den `Browsernamen`, seine `Version` und das verwendete Betriebssystem.

```
alert(navigator.userAgent);
```

- ❑ Die Property `plugins` liefert die im Browser aktuell installierten Plugins (wird nicht von allen Browsern unterstützt)

```
alert(navigator.plugins);
```

- ❑ Cookie-Status: Enabled oder Disabled

```
alert('Status of Cookie Enabled: ' +  
      navigator.cookieEnabled);
```

- ❑ Die vom Browser `unterstützte Sprache` oder `Sprachen` können Sie beispielsweise wie folgt ausgeben (read-only):

```
//Default Sprache "de"  
alert(navigator.language);  
  
//Alle unterstützte Sprachen "de", "en"  
let mylength= navigator.languages.length;  
for(let i=0; i< mylength; i++) {  
    alert("Alle Sprachen: " + i + ': ' +  
          navigator.languages[i] + '\r\n');  
}
```

# navigator-Object

□ Das `navigator`-Object bietet auch die Möglichkeit die Geolocation des Webbrowsers auszulesen. Dazu können Sie wie folgt vorgehen:

- (1) Prüfen Sie, ob Geolocation vom Browser unterstützt wird.
- (2) Führen Sie, falls unterstützt, die Methode `getCurrentPosition()` aus.
- (3) Wenn die Methode `getCurrentPosition()` erfolgreich ist, gibt sie ein Koordinatenobjekt (`position`) an die im Parameter angegebene Callback-Funktion (`showPosition`) zurück.
- (4) Die Funktion `showPosition()` gibt den Breiten- und Längengrad aus

```
if (navigator.geolocation) {  
    navigator.geolocation.getCurrentPosition(showPosition);  
} else {  
    document.getElementById("geo").innerHTML =  
        "Geolocation is not supported."  
}  
  
function showPosition(position) {  
    document.getElementById("geo").innerHTML =  
        "Latitude: " + position.coords.latitude +  
        "Longitude: " + position.coords.longitude;  
}
```

callback

# localStorage-Objekt

- ❑ Das `localStorage`-Objekt ermöglicht das **lokale** Speichern von **Schlüssel/Wert-Paaren** im Browser für eine Webseite (Applikation).
- ❑ Das `localStorage`-Objekt speichert die Daten **lokal**, **permanent** und **verschlüsselt** im Filesystem des lokalen Gerätes.
- ❑ Beispiel: Google Chrome

"C:\Users\...\AppData\Local\Google\Chrome\User Data\Default\Local Storage"
- ❑ Die Daten werden beim Schließen des Browsers nicht gelöscht und stehen für zukünftige Sitzungen zur Verfügung.

## Application

- Manifest
- Service Workers
- Storage

## Storage

- Local storage
- https://www.w3schools.com/**

Applikation

Filter

Key	Value
Paul	123
<b>Alice</b>	<b>12jdh46dhs</b>

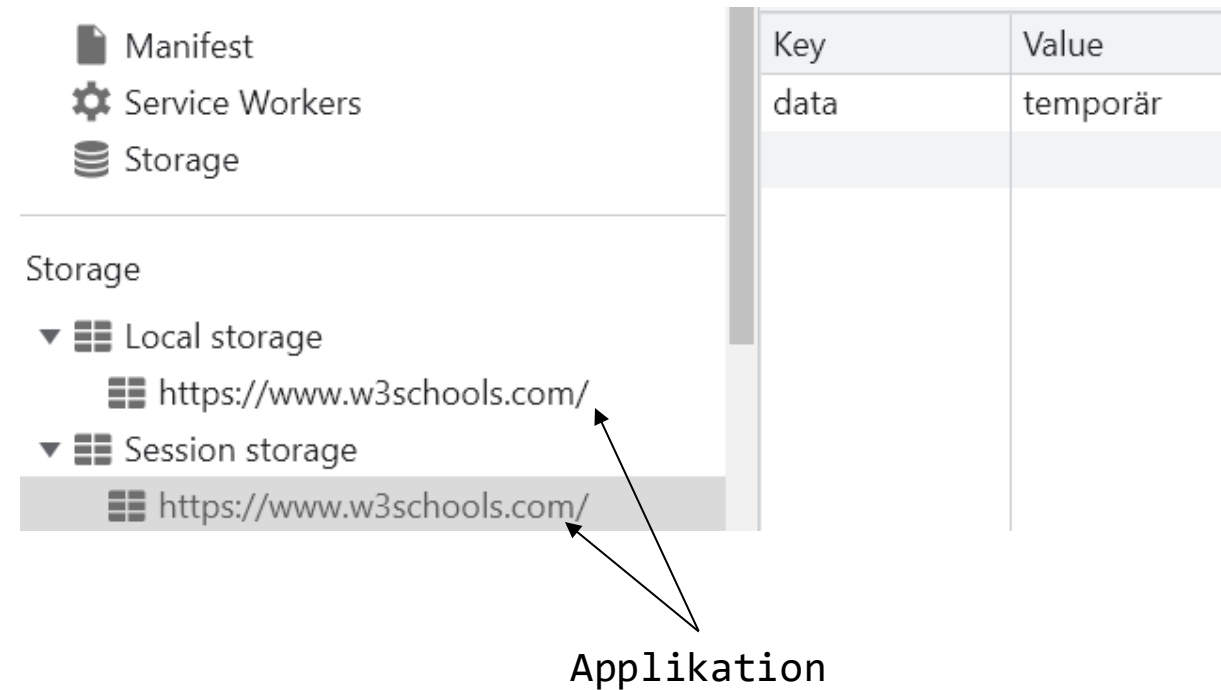
```
//Daten im localeStorage speichern
localStorage.setItem("Alice", "12jdh46dhs");
//Auslesen von Daten mittels des Key-Wertes
let password = localStorage.getItem("Alice");
//Daten im localStorage löschen
localStorage.removeItem('Alice');
```

# sessionStorage-Objekt

- ❑ Das `sessionStorage`-Objekt ermöglicht das **lokale temporäre** Speichern von **Schlüssel/Wert-Paaren** im Browser für eine **Sitzung**.
- ❑ Die Daten werden ebenfalls **verschlüsselt** im Filesystem des lokalen Gerätes.
- ❑ Beispiel: Google Chrome

"C:\Users\...\AppData\Local\Google\Chrome\User Data\Default\Session Storage"

- ❑ Die Daten werden beim Schließen des Browsers **gelöscht** und stehen für zukünftige Sitzungen **nicht** zur Verfügung.



```
//Daten im sessionStorage speichern  
sessionStorage.setItem("data", "temporär");  
//Auslesen von Daten mittels des Key-Wertes  
let content = sessionStorage.getItem("data");
```

# history- und screen-Object

- ❑ Das **history-Objekt** enthält die vom Benutzer besuchten URLs (im Browserfenster).
- ❑ Das Objekt ermöglicht ein **Back-** und **Forward-Button** auf einer Webseite zu platzieren.

```
//Forward Button in browsing history
btnFor= document.getElementById('forward');
btnFor.addEventListener('click',
    function() { history.forward(); },
    false);

//Backward Button in browsing history
btnBack= document.getElementById('backButton');
btnBack.addEventListener('click',
    function() { history.back(); },
    false);
```

- ❑ Das **screen-Objekt** enthält Informationen über den Bildschirm des Besuchers.
- ❑ Folgende Bildschirmereigenschaften können abgefragt werden. Die Größenangaben erfolgen in Pixel (px):

```
//Breite b des phys. Bildschirms in Pixel
let b = screen.width;

//Farbtiefe: Anzahl der Bits pro Pixel
let farbTiefe = screen.colorDepth;

//Verfügbare Höhe für die Webseite (ohne
//Taskbar, ...)
let myHeight = screen.availHeight;
```

## Aufgabe 9: JavaScript

- (1) Viele Browser stellen eine sogenannte Console zur Verfügung. Was versteht man unter dem Begriff REPL? Erläutern Sie dieses an einem konkreten JavaScript-Beispiel.
- (2) Erstellen Sie ein JavaScript, dass die Primzahlen der folgenden 13-stelligen Dezimalzahl 7008514751431 bestimmt und die Zeit zur Berechnung dieses **Faktorisierungsproblem** misst. Erhöhen Sie die Anzahl der Dezimalstellen und vergleichen Sie die Rechenzeiten.  
Führen Sie das Script in der Console ihres Browsers aus.
- (3) Das Verschlüsselungsverfahren RSA verwendet 3072 Bits für das Faktorisierungsproblem. Wieviel Stellen besitzt die zugehörige Dezimalzahl?
- (4) Erstellen Sie nun eine HTML-Seite mit einem `<body>` ohne Inhalt. Ergänzen Sie den `<body>` per JavaScript mit einem Button (Text: "Finde meine Geokoordinaten", Farbe: "orange", keine Umrandung, `id="btn1"`), der per Mausklick ihre Geokoordinaten bestimmt und diese im Textfeld des `<body>` ausgibt.
- (5) Erstellen Sie nun ein **modulares JavaScript-Programm**. In der Datei **mathUtils.js** stellen Sie mathematische Funktionen bereit (Addition, Subtraktion, Multiplikation), in der Datei `constants.js` definieren Sie konstante Größen (*Pi, Eulerzahl, ...*), in der Funktion `main.js` führen Sie dann Berechnungen durch und bringen Sie auf einer Webseite zur Anzeige.