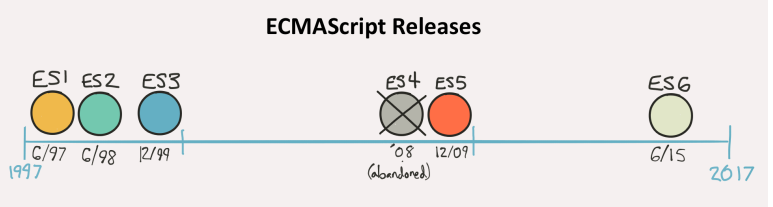
JavaScript

1. JavaScript Entwicklung

1.1. Versionen



https://en.wikipedia.org/wiki/Timeline_of_web_browsers

1.2. Strict mode (1/2)

Mit ECMAScript 5 wurde der **strict mode** eingeführt.

Alle aktuelle Web-Browser (inkl. IE10) implementieren dies.

In Skripten wird dadruch aktiviert, in dem man am Anfang des Quellcodes schreibt:

```
"use strict";
```

1.3. Strict mode (1/2)

Wird "use strict"; hinzugefügt, dann wird in den folgenden Fällen ein SyntaxError ausgelöst, bevor das Script ausgeführt wird:

- Octal-Syntax für Integer: `var n = 023;`
- Statement `with`
- delete mit einem Variablennamen: delete myVariable;
- eval oder arguments als Variable or Name eines Funktionsarguments
- Benutzen der neuen reservierten Schlüsselworte (im Vorgriff auf for ECMAScript 2015): implements, interface, let, package, private, protected, public, static und yield

- Funktions-Deklaration innerhalb von Anweisungsblöcken: `if(a<b){ function f(){} }`
- Offensichtliche (Programmier-) Fehler
  - Mehrfache Deklaration von Properties mit demselben Namen in einem Objektliteral: `{a: 1, b: 3, a: 7}`. Das triff auf ECMAScript 6 nicht länger zu (siehe Bug 1041128).
  - Deklaration mehrerer Funktionsargumente mit dem selben Namen: function f(a, b, b){}

1.4. JavaScript einbetten in HTML-Seite

```
<!DOCTYPE HTML>
<html>

<body>

  <p>Before the script...</p>

  <script>
    alert( 'Hello, world!' );
  </script>

  <p>...After the script.</p>

</body>

</html>
```

1.5. JavaScript referenzieren

```
<script src="/path/to/script.js"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/
lodash.js"></script>
[source,html]
```

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
```

Hinweis: Früher hat man die Attribute The `type` und `language` eingesetzt, das ist nicht mehr nötig.

2. Grundlegendes

2.1. Kommentare

```
// This is a single-line comment.

/* This is also a comment */ // and here is another comment.

/*
   This is yet another comment.
 *
 * It has multiple lines.
 */
```

2.2. Literal

Ein Literal ist ein Wert, der direkt im Programm vorkommen kann. Folgende Typen gibt es:

```
12        // The number twelve
1.2       // The number one point two
"tutego" // A string of text
'Hi'      // Another string
true      // A Boolean value
false     // The other Boolean value
/js/gi    // A "regular expression" literal (for pattern matching)
null      // Absence of an object
```

2.3. Bezeichner (Identifier)

Ein JavaScript Bezeichner muss mit einem Buchstaben, einem Unterstrich _ oder einem Dollar-Zeichen beginnen.

Danach können auch Ziffern folgen.

Gültig:

```
i
```

```
my_variable_name
v13
_dummy
$str
```

Tipp: Nutze englischsprachige Bezeichner.

2.4. Variablennamen

Verwendet sprechende Namen, d. h. sinnvolle Bezeichnungen, die sich später zurückverfolgen lassen.

```
ausgabe = breite * höhe;
```

ist besser als

```
a = b * h;
```

Variablen sind case-sensitive! Daher ist auf Groß-und Kleinschreibung zu achten.

Üblich ist ist das so genannte **CamelCase**, in dem die ersten Buchstaben der einzelnen Bestandteile des Namens groß geschrieben werden.

2.5. Reservierte Schlüsselwörter

Reservierte Schlüsselwörter in ECMAScript 2015

```
break case catch class const continue debugger default delete do
else export extends finally for function if import in instanceof new
return super switch this throw try typeof var void while with yield
```

Reservierte zukünftige Schlüsselworte

```
enum
```

strict mode

```
enum implements interface let await
```

https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Lexical_grammar

3. Tyen, Werte, Variablen, Ausdrücke, Operatoren

3.1. Zahlen

- Es gibt in JavaScript keinen Unterschied zwischen Fließkommazahlen und Ganzzahlen.
- Speicherung nach IEEE 754.
- Ganzahlen sind ohne Rundungsfehler im Bereich −9007199254740992 bis 9007199254740992 möglich.
- Stehen Zahlen direkt im Code nennt man das **numeric literal**. Es gibt mehrere Schreibqweisen
  - Beginnen nummerische Literale mit `0x` oder `0X` leitet das Hexadezimalzahlen ein.
  - Fließkommzahlen mit Exponenten: `[digits][.digits][(E|e)[(+|-)]digits]`

3.2. Besondere symbolische Werte

Neben Gleitkommazahlen kann der Datentyp für Zahlen auch drei symbolische Werte annehmen:

- +Infinity
- -Infinity
- NaN

`0` kann als `-0` und `+0` repräsentiert werden. Und es ist `+0 === -0` gleich `true`. Aber:

```
42 / +0    // Infinity
42 / -0    // -Infinity
```

3.3. String für Text

Beispiel für **String-Literale**:

```
""      // The empty string: it has zero characters
'testing'
```

```
"3.14"
'name="myform"'
"Wouldn't you prefer Christian's book?"
"This string\nhas two lines"
"π is the ratio of a circle's circumference to its diameter"
```

3.4. Escape Sequences

```
\0       The NUL character (\u0000)
\b       Backspace (\u0008)
\t       Horizontal tab (\u0009)
\n       Newline (\u000A)
\v       Vertical tab (\u000B)
\f       Form feed (\u000C)
\r       Carriage return (\u000D)
\"       Double quote (\u0022)
\'       Apostrophe or single quote (\u0027)
\\       Backslash (\u005C)
\x XX    The Latin-1 character specified by the two hexadecimal digits
 XX
\u XXXX  The Unicode character specified by the four hexadecimal digits
 XXXX
```

3.5. Operatoren

Es gibt in JavaScript die typischen Operatoren, *, /, +, -.

Mit Klammen kann man den Vorrang setzen.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

3.6. Konkatenieren von Strings mit

```
msg = "Hello, " + "world"; // Produces the string "Hello, world"
greeting = "Welcome to my blog," + " " + name;
```

3.7. Boolean Werte

- Es gibt die Literale `true` und `false`.
- Das Ergebnis von Vergleichen ist vom Typ Boolean.

3.8. Variablendeklaration

Bevor man in JavaScript Variablen nutzt, *sollten* sie deklariert werden. (Im strikten Modus ist das auch zwingend. Wir machen das so!)

Dazu gibt es drei Schlüsselworte:

- `let`: Deklariert Variable im aktuellen Block
- `const`: Deklariete Konstante
- `var`: Deklariert eine Variable (veraltet)

3.9. Beispiel für Variablendeklaration

```
let message;
let greeting = "hello";
let i = 0, j = 0, k = 0;
const PI = 3.14159265;
```

3.10. Dynamische Typisierung

Eine Variable enthält in JavaScript irgendwelche Daten. Die Datentypen können sich jederzeit ändern.

```
let id = "4711";
id = 4711;
```

Der Datentyp keine Eigenschaft einer Variablen, sondern Sache der Laufzeitumgebung.

- JavaScript ist einne **schwach typisierte** oder **dynamische Programmiersprache**.

Es gibt Erweiterungen von JavaScript, wie **TypeScript**, die eine statische Typisierung ermöglichen.

3.11. Variablentypen

JavaScript unterscheidet:

6 Primitive Datentypen

- boolean
- null
- undefined
- number
- string
- symbol

Nicht primitive Datentypen

- Objekte
- Funktionen
- reguläre Ausdrücke
- Arrays
- …

Alle Datentypen, bis auf `Object`, definieren unveränderbare Werte.

3.12. Undefined Datentyp

Eine Variable, der noch kein Wert zugewiesen wurde, hat den Wert `undefined`.

```
var a;
console.log("a ist " + a); // a ist undefined
```

Das ist ein primitiver Wert, der automatisch gerade erst deklarierten Variablen zugewiesen wird.

Es ist auch ein Wert, den angegebene Argumente annehmen, wenn sie im Funktionsaufruf nicht verwendet werden.

Tipp: `undefined` kann benutzt werden, um zu überprüfen ob eine Variable einen Wert hat: `if(input === undefined) … .`

3.13. ReferenceError

Unter der Annahme das wir im strict-Mode arbeiten: Greifen wir

- auf eine Variable zu, die nicht existiert, oder
- auf `let`-Variable zu, bevor diese initialisiert wurde,

folgt ein `ReferenceError`.

```
console.log("y ist " + y); // ReferenceError: Cannot access 'y' before
 initialization
let y;

let z;
console.log("z ist " + z); // z ist undefined
```

3.14. typeof Operator

Der Operator `typeof` gibt einen String zurück, der den Typ des unausgewerteten Operanden beschreibt.

Typ	Rückgabewert
Zeichenkette	"string"
Zahl	"number"
`true` oder `false`	"boolean"
Undefined	"undefined"
`null`	"object"
`Symbol`	"symbol"
Funktionsobjekt	"function"
Alle anderen Objekte	"object"
Host-Objekt	implementierungsabhängig

Beispiele siehe https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Operators/typeof

3.15. Implizite Typkonvertierungen

JavaScript ist sehr "flexibel" wenn es um die Umwandlungen von Typen geht.

- Wenn JavaScript einen Typ "will" konvertiert er in in diesen. Die Regeln sind nicht immer offensichtlich.

- Bei einem `alert(… )` wird zum Beispiel in ein String konvertiert, bei mathematischen Operationen in Zahlwerte.

```
10 + " Elfen" // => "10 Elfen". Zahl 10 wird zum String
"7" * "4" // => 28: beide Stings werden zu Zahlen
var n = 1 - "x"; // => NaN: String "x" kann nicht in eine Zahl
 konvertiert werden
n + " objects" // => "NaN objects": NaN wird zum String "NaN"
```

3.16. Falsy Werte

Bei Fallunterscheidungen und Schleifen sind Bedingungen nötig, die Wahrheitswerte sind.

Die folgenden Werte werden zu `false` ausgewertet (auch bekannt als **Falsy Werte**):

- false
- undefined
- null
- 0
- NaN
- der leere String ("")

Alle anderen Werte, auch alle Objekte, werden zu `true` ausgewertet, wenn sie als Bedingung eingesetzt werden.

3.17. Explizite Typkonvertierungen

JavaScript führt viele Typkonvertierungen automatisch durch. Wir können das auch explizit schreiben:

Ein Weg ist über die die Funktionen `Boolean()`, `Number()`, `String()` oder `Object()` functions.

```
Number("3")    // 3
String(false)  // "false"
```

```
Boolean([])    // true
Object(3)      // new Number(3)
```

Es gibt clevere Abkürzungen:

```
x + ""         // Wie String(x)
+x             // Wie Number(x)
x-0            // Wie Number(x)
!!x            // Wie Boolean(x)
```

3.18. Vergleichsoperatoren

Operator	**Bedeutung**	**Ergebnis der Operation**
==	ist gleich	true, wenn die Werte gleich sind, sonst false
!=	ungleich	true, wenn die Werte ungleich sind, sonst false
>	größer	true, wenn der linke Wert größer als der rechte ist, sonst false.
>=	größergleich	true, wenn der linke Wert größer oder gleich dem rechten ist, sonst false
<	kleiner	true, wenn der linke Wert kleiner als der rechte ist , sonst false
⇐	kleinergleich	true, wenn der linke Wert kleiner oder gleich dem rechten ist, sonst false

3.19. Typgenaue Vergleiche

Die genannten Vergleichsoperatoren führen automatisch Typkonvertierungen durch.

Um bei Vergleichen auch den Datentyp zu berücksichtigen, gibt es zwei typgenaue Vergleichsoperatoren:

===

`true`, wenn beide Werte vom gleichen Typ und gleich sind, sonst `false`. Auch genannt Strict Equality Comparison ("strict equality", "identity", "triple equals").

!==

`true`, wenn beide Werte vom gleichen Typ und nicht gleich sind, sonst `false`.

Die herkömmlichen Vergleiche heißen im Englischen auch Abstract Equality Comparison ("loose equality", "double equals").

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

4. Anweisungen

4.1. Block-Anweisungen

Eine Block-Anweisung gruppiert Anweisungen. Dieser Block wird von einem Paar geschweiften Klammer umschlossen:

```
{
  statement_1;
  statement_2;
  .
  .
  .
  statement_n;
}
```

4.2. Redeklarationen

Eine erneute Deklaration derselben Variable innerhalb desselben Gültigkeitsbereiches erzeugt einen Syntaxfehler (`SyntaxError`).

```
{
  let foo;
  let foo; // SyntaxError thrown.
}
```

Achtung: Der Körper einer `switch`-Anweisung ist nur ein einzelner Block. Allerdings kann man mit `{}` wieder einen eigenen Block einführen.

5. Gültigkeitsbereich: Globale oder Funktion

Deklariert man Variablen außerhalb von Funktionen spricht man von **globalen Variablen**.

- Sie sind in einem globalen Gültigkeitsbereich und man kann überall auf sie zugreifen.

Daneben gibt es den **Funktions-Gültigkeitsbereich**.

- Die Variablen sind nur in Funktionen gültig. Hier können auch **lokale** Variablen deklariert werden.

5.1. Gültigkeitsbereich (Variable Scope)

Es gibt einen wichtigen Unterschied zwischen `let` und `var`:

- Variablen, die mit `let` deklariert werden, haben als Gültigkeitsbereich den Block in dem sie definiert wurden und alle weiteren Unterblöcke in denen sie nicht neu definiert werden.
- Der Gültigkeitbereich bei `var`-deklarierten-Variablen ist die umschließende Funktion (oder für Deklarationen außerhalb von Funktionen der globale Kontext).

5.2. var hoisting (1/2)

Bei `var` werden Variablendeklarationen immer ausgeführt bevor Programmcode ausgeführt wird, egal wo sie im Programmcode vorkommen.

Man spricht von **hoisting**, wenn die Deklaration der Variablen **an den Anfang der Funktion** oder den **Anfang des globalen Programmcodes** verschoben wird.

Es ist also identisch:

```
bla = 2; var bla;
```

und

```
var bla; bla = 2; var bla; ⟶ var bla; bla = 2;
```

5.3. var hoisting (2/2)

```
var a = 1;

function four() {
  if (true) {
    var a = 4;
  }

  alert(a); // alerts '4', nicht '1'
}
four();
```

Tipp: Definiere Variablen immer am Anfang ihres Gültigkeitsbereiches.

5.4. Was ist die Ausgabe?

```
{
  let food = 'Meow Mix';
}

console.log(food);
```

Was passiert, wenn man `let` auf `var` ändert?

5.5. Aufgabe: Was ist die Ausgabe? Was ergibt die Änderung var → let?

```
var snack = 'Meow Mix';

function getFood(food) {
    if (false) {
        var snack = 'Friskies';
        return snack;
    }
    return snack;
}

getFood();
```

5.6. Bedingte Anweisungen

```
if (bedingung_1) {
  statement_1;
} else if (bedingung_2) {
  statement_2;
} else if (bedingung_n) {
  statement_n;
} else {
  statement_last;
}
```

5.7. switch-Anweisung

```
switch (ausdruck) {
  case label_1:
    statements_1
    [break;]
  case label_2:
    statements_2
    [break;]
    ...
  default:
    statements_def
    [break;]
}
```

5.8. while-Schleife

```
while (bedingung)
  anweisung;
```

5.9. do-while-Schleife

```
do {
  anweisung
} while (bedingung);
```

5.10. for-Schleife

```
for (startausdruck; bedingung; iterationsausdruck)
    anweisung
```

5.11. Fehler- bzw. Ausnahmebehandlung

Man kann mit

- `throw` Ausnahmen werfen (erzeugen) und
- diese mit `try-catch` Statements auffangen und verarbeiten und
- optional mit einem `finally` Block Nachbararbeitungen durchführen.

5.12. Beispiel für thow und try-catch

Man kann jeden Ausdruck werfen, nicht nur Ausdrücke eines bestimmten Typs.

```
throw "Error2";   // String type
```

Auffangen:

```
try {
  throw "myException" // Erstellt eine Exception
}
catch (e) {
  // Statements, die die Exception verarbeiten
}
```

6. Funktionen

6.1. Funktionsdefinition

Eine Funktionsdefinition (auch Funktionsdeklaration oder Funktionsanweisung genannt) besteht aus dem Schlüsselwort `function`, gefolgt von:

- Dem Namen der Funktion.
- Eine Liste von Parametern, die in runden Klammern geschrieben sind und durch Kommas getrennt sind.
- Einem Rumpf: Anweisungen, die durch die Funktion definiert werden und innerhalb von geschweiften Klammern stehen.

```
function square(number) {
  return number * number;
}
```

6.2. Argumentübergabe: Call-by-Value

Bei **primitiven Typen**, wie Zahlen, oder Strings, wird der Funktionen der Wert übergeben.

Werte, die der Funktion übergeben wurden und innerhalb der Funktion geändert werden, ändert den Wert zwar innerhalb der Funktion, aber nicht global oder in der aufrufenden Funktion.

Anders ist das bei nicht-primitiven Typen, wie Arrays.

6.3. Funktionsausdrücke

Die Funktionsdeklarationen kann ein Ausdruck sein oder auch durch eine **Funktionsausdruck** erstellt werden.

Derartige Funktionen können auch anonym sein; denn Funktionen benötigten keinen Namen. So kann zum Beispiel die Funktion `square` auch so definiert werden:

```
var square = function(number) { return number * number; };
var x = square(4); // x bekommt den Wert 16
```

6.4. Funktion sind Objekte

Funktionen sind in JavaScript vollwertige Objekte.

- Sie haben Methoden und Eigenschaften, können erstellt und überschrieben, als Argumente an Funktionen übergeben und von ihnen erzeugt und zurückgegeben werden.

Funktionsausdrücke sind praktisch, um Funktionen als ein Argument einer anderen Funktion zu übergeben.

7. Arrays

Ein Array erstellen:

```
var names  = [];
var fruits = ['Apple', 'Banana'];
```

Länge vom Array:

```
console.log(fruits.length);  // 2
```

Die Länge ist auch beschreibar. So kann man das Array kürzen oder "nichts" auffüllen.

Zugriff auf ein Arrayelement (mit Index)

```
var first = fruits[0];                  // Apple

var last = fruits[fruits.length - 1];   // Banana
```

Arrays sind Objekte, bei Objekten ist die Zugriffsart die gleiche.

8. Methoden von Arrays (1/3)

Ein Element am Ende des Arrays einfügen:

```
var newLength = fruits.push('Orange');
// ["Apple", "Banana", "Orange"]
```

```
fruits[4] = 'Lemon';
// "Apple", "Banana", "Orange", empty, "Lemon"]
```

Ein Element am Ende des Arrays löschen:

```
var last = fruits.pop(); // remove Orange (from the end)
// ["Apple", "Banana"];
```

Ein Element am Anfang des Arrays löschen:

```
var first = fruits.shift(); // remove Apple from the front
// ["Banana"];
```

9. Methoden von Arrays (2/3)

Ein Element am Anfang des Arrays einfügen:

```
var newLength = fruits.unshift('Strawberry')
// ["Strawberry", "Banana"];
```

Den Index eines Elements im Array ermitteln:

```
fruits.push('Mango');  // ["Strawberry", "Banana", "Mango"]

var pos = fruits.indexOf('Banana'); // 1
```

Ein Array kopieren

```
var shallowCopy = fruits.slice(); // this is how to make a copy
// ["Strawberry", "Mango"]
```

10. Methoden von Arrays (3/3)

Ein Element mithilfe eines Index aus dem Arrays löschen:

```
var removedItem = fruits.splice(pos, 1);

// ["Strawberry", "Mango"]
```

Elemente von einer Indexposition aus löschen

```
var vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];
console.log(vegetables); // ["Cabbage", "Turnip", "Radish", "Carrot"]

var pos = 1, n = 2;

var removedItems = vegetables.splice(pos, n);
// this is how to remove items, n defines the number of items to be
 removed,
// from that position(pos) onward to the end of array.

console.log(vegetables);
// ["Cabbage", "Carrot"] (the original array is changed)

console.log(removedItems);
// ["Turnip", "Radish"]
```

https://www.ecma-international.org/ecma-262/7.0/#sec-array-objects

10.1. Schleifen über Arrays (1/2)

Klassisch:

```
let arr = ["Apple", "Orange", "Pear"];
for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

Moderner mit for-of:

```
let fruits = ["Apple", "Orange", "Plum"];
for (let fruit of fruits) {
  alert( fruit );
}
```

Achtung: Es gibt auch ein `for-in`, doch das macht etwas anderes!

10.2. Schleifen über Arrays (2/2)

```
var txt = "";
```

```
let fruits = ["Apple", "Orange", "Plum"];
fruits.forEach(myFunction);

function myFunction(value, index, array) {
  txt = txt + value + "<br>";
}
```

Weitere besondere Array-Funktionen: https://www.w3schools.com/js/js_array_iteration.asp

11. Wrapper Objekte und die Standardbibliothek