JavaScript
Datentypen und Variablen

Eigenschaften von JavaScript

- C ähnliche Syntax, frei formatierbar
- interpretierend
- schwaches Typsystem (loosely-typed)
- Garbage-Collection ähnlich zu Java
- Strings in Unicode
- objektbasiert aber nicht objektorientiert
  - es gibt keine Klassen
  - Objekte haben Eigenschaften und Methoden
- Variablen und Funktionen können jederzeit angelegt werden
- variable Anzahl von Parametern für Funktionen
- portabel und hardwareunabhängig

JavaScript vs. Java

- ähnliche Syntax:
  - Blöcke mit `{ }`
  - Kontrollstrukturen: `if, switch, for, while, do, try catch`
  - Kommentare: `//, /*...*/`
  - Stringverkettung mit `+`
- JavaScript hat keine feste Typdeklaration:
  - eine Variable kann nacheinander unterschiedliche Typen halten.
  - Zahlen sind immer Gleitkommazahlen (kein int, kein short, ...).
  - `function` leitet eine Funktionsdeklaration ein.

Variablen

- Zur Erstellung von Variablen können drei verschiedene Identifikatoren benutzt werden.
- Angelegte, aber nicht initialisierte Variablen sind automatisch **`undefined`**.
- Der Typ einer Variablen kann mittels **`typeof`** abgefragt werden.
- Identifikatoren sind:
  - **`var`**
  - **`let`**
  - **`const`**

var versus let in Funktionen

- 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.
  - In dieser Hinsicht funktioniert let ähnlich wie var.
- Der Unterschied zum var Schlüsselwort ist, dass der Gültigkeitsbereich auf Blöcke und nicht auf Funktionen bzw. Global beschränkt ist.

```
function varTest() {
  var x = 31;
  if (true) {
    var x = 71;  // gleiche Variable!
    console.log(x);  // 71
  }
  console.log(x);  // 71
}
```

```
function letTest() {
  let x = 31;
  if (true) {
    let x = 71;  // andere variable
    console.log(x);  // 71
  }
  console.log(x);  // 31
}
```

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

var versus let in Objekten

- Auf der ersten Ebene von Programmen und Funktionen erzeugt let im globalen Objekt keine Eigenschaft, var hingegen schon...

```
var x = 'global';
let y = 'global';
console.log(this.x); // "global"
console.log(this.y); // undefined
```

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

Datentypen

- Eingebaute Datentypen von JavaScript:
  - **`Boolean`** – Wahrheitswert (true oder false)
  - **`Number`** – Ganz- oder Fließkommazahl
  - **`String`** – Zeichenkette
  - **`Date`** – Datum
  - **`Array`** – Indiziertes oder assoziatives Array
  - **`Objekt`** – Vor- oder selbstdefinierte Objekte
  - **`Function`** – Funktionen
  - **`RegExp`** – Reguläre Ausdrücke

Literale und Instanzen

- Die meisten primitiven Datentypen können auf zwei Arten erzeugt werden:
  - über Literale oder
  - über Instanzen.
- Die Datentypen **`Number`**, **`Boolean`**, **`String`** und **`Object`** können über den Aufruf eines Konstruktors erzeugt werden.
- Die Konstruktor-Funktionen heißen genau wie die Datentypen und liefern ein Objekt zurück.
  - Dies gilt für Arrays und reguläre Ausdrücke.
- Für **`Null`** und **`Undefined`** existieren nur Literale.

Literale und Instanzen

- String:
  - Literal: `"aaa" / 'aaa'`
  - Konstruktor: `new String(value)`
- Number:
  - Literal: `123, ...`
  - Konstruktor: `new Number(value)`
- Boolean:
  - Literal: `true / false`
  - Konstruktor: `new Boolean(value)`
- Object:
  - Literal: `{}`
  - Konstruktor: `new Object()`

Typumwandlung

- die Typumwandlung erfolgt implizit
  - hin zum größeren Typ
  - keine Genauigkeitsverluste bei Zuweisung

- es gibt 2 Vergleichsoperatoren
  - `==` konvertiert die Werte erst zu einem gemeinsamen Typ und vergleicht sie dann.
  - `===` vergleicht die Werte sofort, ohne eine Typangleichung durchzuführen. Bei ungleichem Typ ist der Vergleich dadurch immer negativ.
  - Der „triple equals“-Operator ist immer zu bevorzugen!
  - Die Benutzung von „double equals“ kann zu übersehenen Bugs führen!

- `typeof(variable)` liefert den Datentyp der Variablen

Typumwandlung: Beispiel

```
var a = "10";
var b = 10;
document.write("typeof(a) = " + typeof(a) + "<br/>");
document.write("typeof(b) = " + typeof(b) + "<br/>");
var c1 = (a == b);
var c2 = (a === b);
document.write("c1 = " + c1 + "<br/>");
document.write("c2 = " + c2 + "<br/>");
```

typeof(a) = string
typeof(b) = number
c1 = true
c2 = false

Truthiness

- Die Operatoren `||, &&` und Kontrollstrukturen wie `if` arbeiten mit boolschen Werten; beliebige Daten können übergeben werden.
- Die Truthiness legt fest, welche Werte als `true` bzw. `false` angesehen werden.
- falsch sind:
  - `false`
  - `0, -0, NaN`
  - `null, undefined`
- wahr sind alle anderen Werte.

Undefined und Null

- Alle Variablen, die deklariert, aber nicht initialisiert sind, haben automatisch den Wert **`undefined`**.
- Null repräsentiert hingegen ein gewolltes Auslassen einer Wertes
  - Null hat nur einen Wert: **`null`**.

Scope von Variablen

- In JavaScript existieren zwei Arten von Scopes.
- Diese definieren die Sichtbarkeit einer Variable.

- 1. globaler Scope:
  - Alle nicht in einer lokalen Funktion definierten Variable befinden sich im globalen Scope.
  - Dieser beinhaltet abhängig von der Laufzeitumgebung verschiedene APIs und die Standardbibliothek.

Scope von Variablen

- In JavaScript existieren zwei Arten von Scopes.
- Diese definieren die Sichtbarkeit einer Variable.

- 2. lokaler Scope:
  - Jede Funktion definiert einen neuen, lokalen Scope.
  - Alle in einer Funktion definierten Variablen sind von ihrem Scope, aber nicht vom globalen Scope sichtbar.
  - Innerhalb eines lokalen Scopes kann man auf alle äußeren Scopes zugreifen.
  - Ein lokaler Scope wird bei der Ausführung der Funktion erstellt und nach ihrer Beendigung entfernt.
  - Achtung: Falls eine Variable in einem lokalen Scope initialisiert, aber nicht definiert wird, ist sie automatisch global.

Scope von Variablen

- Variablen sind in der ganzen Funktion sichtbar.
- Blöcke haben -anders als in Java- keine Auswirkungen auf die Sichtbarkeit.

```
var a = 5;  // global sichtbar
function f() {
    var b = 7; // innerhalb der Funktion sichtbar
    if (b === 7) {
        var c = 9; // innerhalb der Funktion sichtbar
    }
    document.write("a = " + a +"<br/>");
    document.write("b = " + b +"<br/>");
    document.write("c = " + c +"<br/>");
}
f();
```

a = 5
b = 7
c = 9

Scope von Variablen

```
let globaleVariable = 5;
let variable = 4;
function neuerScope() {
    let variable = '';
    neueGlobaleVariable = 2;
    console.log(typeof variable); // string
    console.log(typeof globaleVariable); // number
    console.log(typeof neueGlobaleVariable); // number
}
console.log(typeof globaleVariable); // number
console.log(typeof variable); // number
console.log(typeof neueGlobaleVariable); // undefined
neuerScope();
console.log(typeof neueGlobaleVariable); // number
```

JavaScript Debug-Konsole in Safari

Web-Entwickler -> Web-Konsole

number	Globaler Code — _display:49
number	Globaler Code — _display:50
undefined	Globaler Code — _display:51
string	neuerScope — _display:45
number	neuerScope — _display:46
number	neuerScope — _display:47
number	Globaler Code — _display:53

Not a Number (NaN)

- bei Zahlen repräsentiert **`NaN`** den ungültigen Wert Not a Number.
- Kein Wert ist gleich **`NaN`**; auch nicht NaN selbst!

```
document.write(1 / 0);
document.write("<br/>");
document.write(0 / 0);
document.write("<br/>");
document.write(0 === NaN);
document.write("<br/>");
document.write(NaN === NaN);
document.write("<br/>");
```

Infinity
NaN
false
false

Einfache Literale

- Notation ist ähnlich zu Java; mit Besonderheiten bei Strings

```
var antwort = 42;       // Integer-Literal
var pi = 3.141;         // Fließkomma-Literal
var happy = true;       // Boolean-Literal
var leer = null;        // Null-Literal

// String-Literale:
var s1 = 'Hallo';
var s2 = "Hallo";
var s3 = 'String "im String" geht auch';
var s4 = "String 'im String' geht auch";
var s5 = "String " + "Anhang";
var s6 = "\"Escapen von Zeichen\"";
var s7 = "\u00ea Unicode";
```

Hoisting

- Mit Hoisting (Hochziehen) bezeichnet man das Verhalten des JavaScript-Interpreters bei der Deklaration von Variablen und Funktionen in Funktionen.
- Stößt der JavaScript-Interpreter auf eine Funktion, so überprüft er zunächst in der gesamten Funktion, welche Variablen im lokalen Gültigkeitsbereich der Funktion definiert werden und welche global definiert wurden.
- Werden Variablendefinitionen mit lokalem Gültigkeitsbereich gefunden, so werden diese gleich deklariert, nicht aber initialisiert.
- Der JavaScript-Interpreter "zieht" also sämtliche Deklarationen von Variablen im Funktionsrumpf gleich unterhalb des Kopfes der Funktion hoch.
- Ebenso werden die Deklaration von Funktionen nach vorne gezogen.

Hoisting

```
var s = "hallo";      // globale Variable
function foo() {
    document.write(s);  // ergibt "undefined" und nicht "hallo“
    var s = "test";     // Initialisierung von 's'
    document.write(s);  // ergibt "test“
}
foo();
```

Der obenstehende Code wird vom JavaScript-Interpreter so ausgeführt, als wäre Folgendes definiert worden...

Hoisting

```
var s = "hallo";      // globale Variable
function foo() {
    var s;
    document.write(s);  // ergibt "undefined" und nicht "hallo“
    s = "test";         // Initialisierung von 's'
    document.write(s);  // ergibt "test“
}
foo();
```