

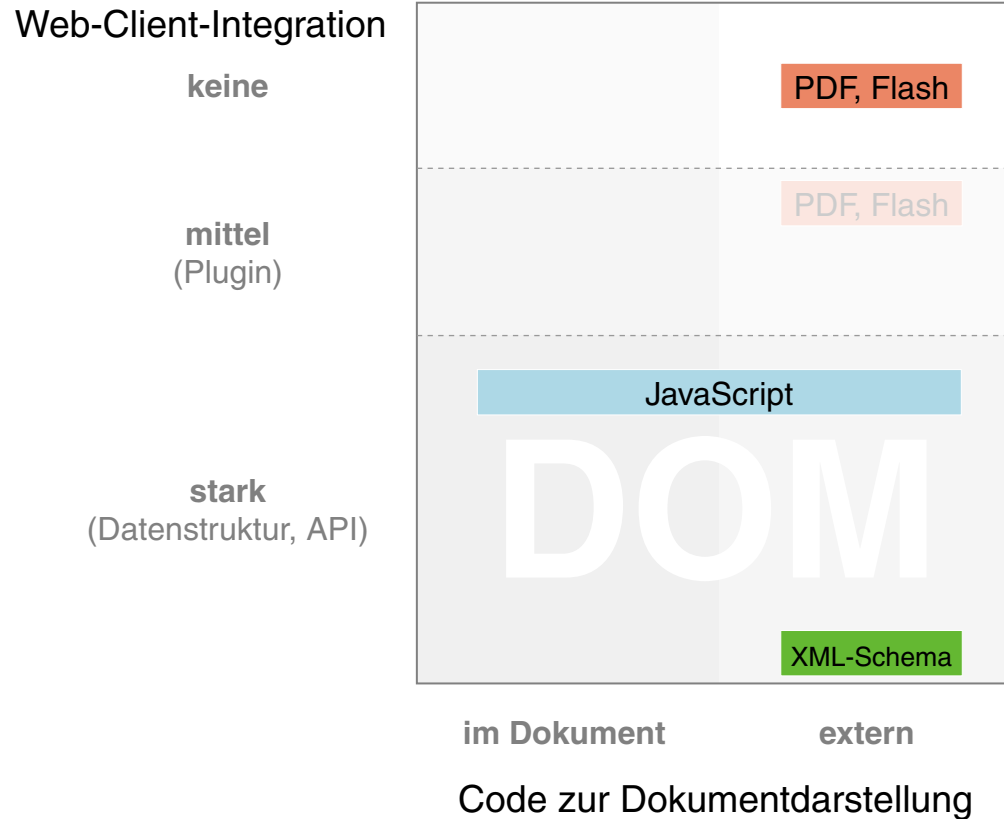
# Kapitel WT:V

## V. Client-Technologien

- ☐ Web-Client
- ☐ Exkurs: Programmiersprachen
- ☐ JavaScript
- ☐ VBScript
- ☐ Java Applet
- ☐ Weitere Client-Technologien

# Web-Client

## Einordnung von Client-Technologien [Stein 2012 - 2020]



- ❑ x-Achse: Wo befindet sich der Code zur Dokumentdarstellung?
- ❑ y-Achse: Wie stark ist die Technologie in den Web-Client integriert?

# Web-Client

## Einordnung von Client-Technologien [Stein 2012 - 2020]

Web-Client-Integration

**keine**

PDF, Flash

Java Webstart

**mittel**  
(Plugin)

PDF, Flash

ActiveX

Java Applet

**stark**  
(Datenstruktur, API)

JavaScript

SVG, WebGL

CSS

DTD

XSLT

XML-Schema

im Dokument

extern

Code zur Dokumentdarstellung

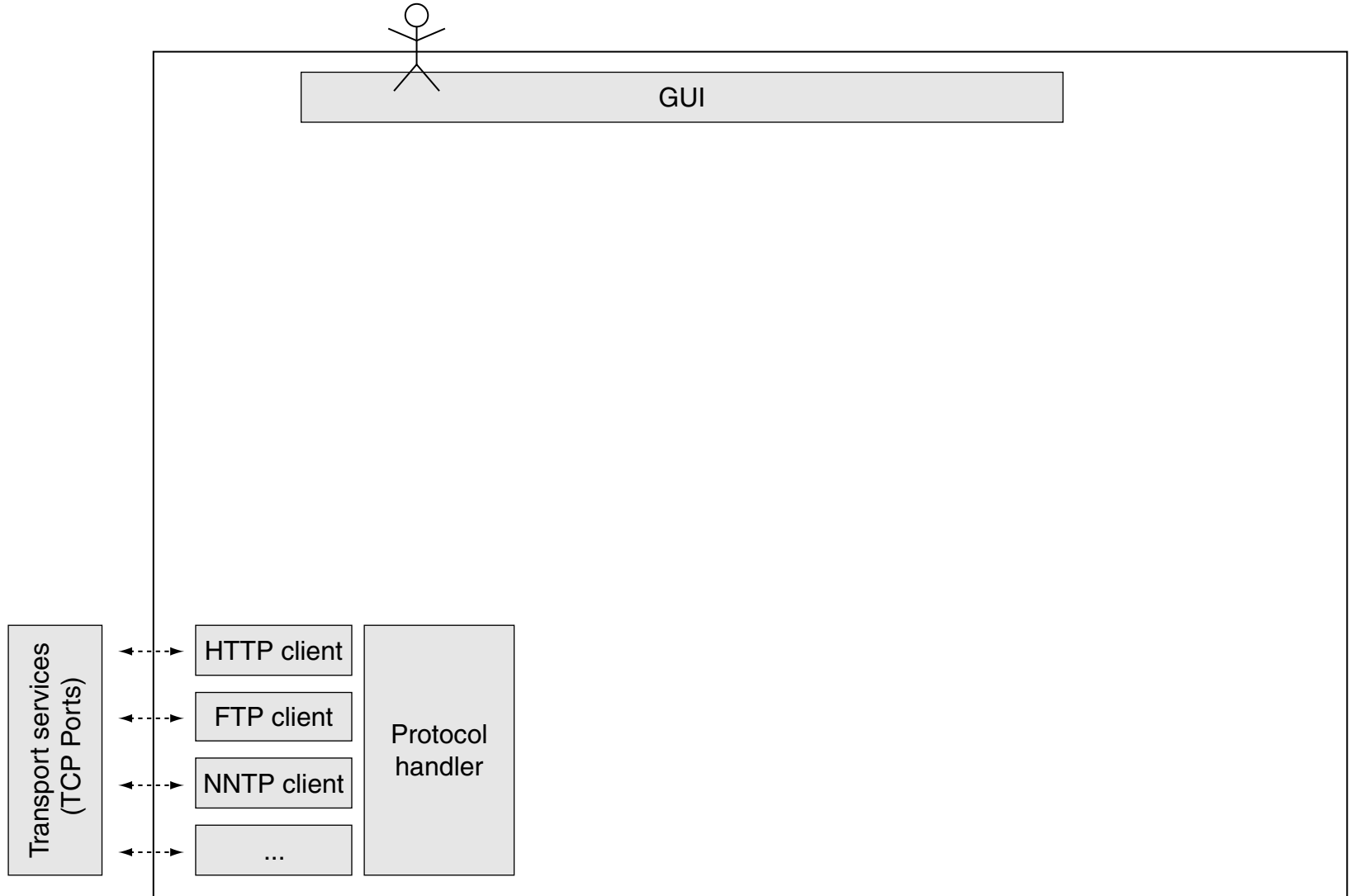
- ❑ x-Achse: Wo befindet sich der Code zur Dokumentdarstellung?
- ❑ y-Achse: Wie stark ist die Technologie in den Web-Client integriert?

## Bemerkungen:

- ❑ Client-Technologien dienen zur Realisierung Client-seitig ablaufender Web-Anwendungen.
- ❑ Im Vergleich zu Server-seitig ablaufenden Web-Anwendungen erzeugen sie weniger Server-Last und mehr Netzlast.

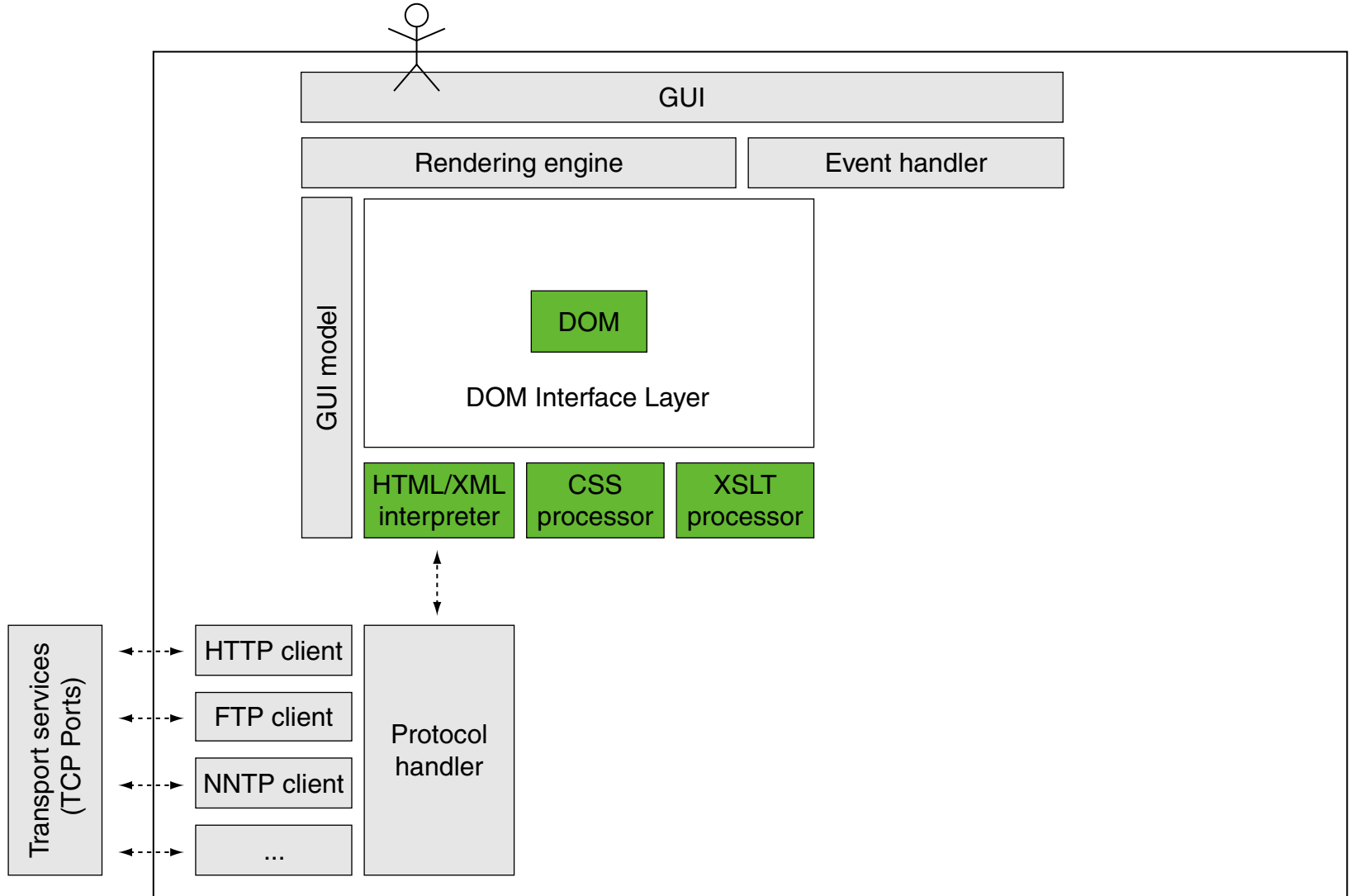
# Web-Client

Browser-Module [MDN [data flow](#)] [W3C [parsing model](#)] [[how browsers work](#)] [Google [1](#), [2](#), [3](#), [4](#)]



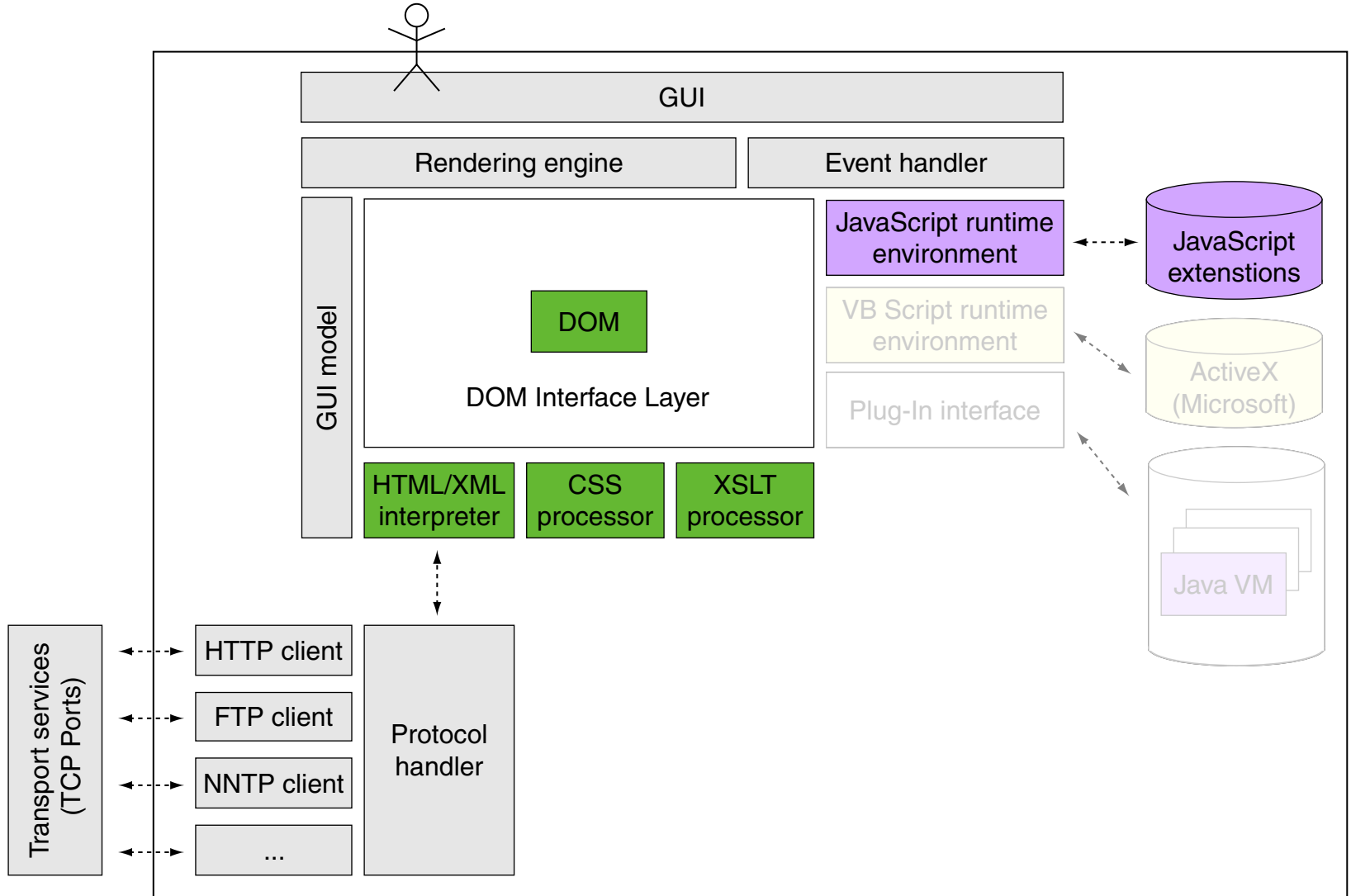
# Web-Client

Browser-Module [MDN [data flow](#)] [W3C [parsing model](#)] [[how browsers work](#)] [Google [1](#), [2](#), [3](#), [4](#)]

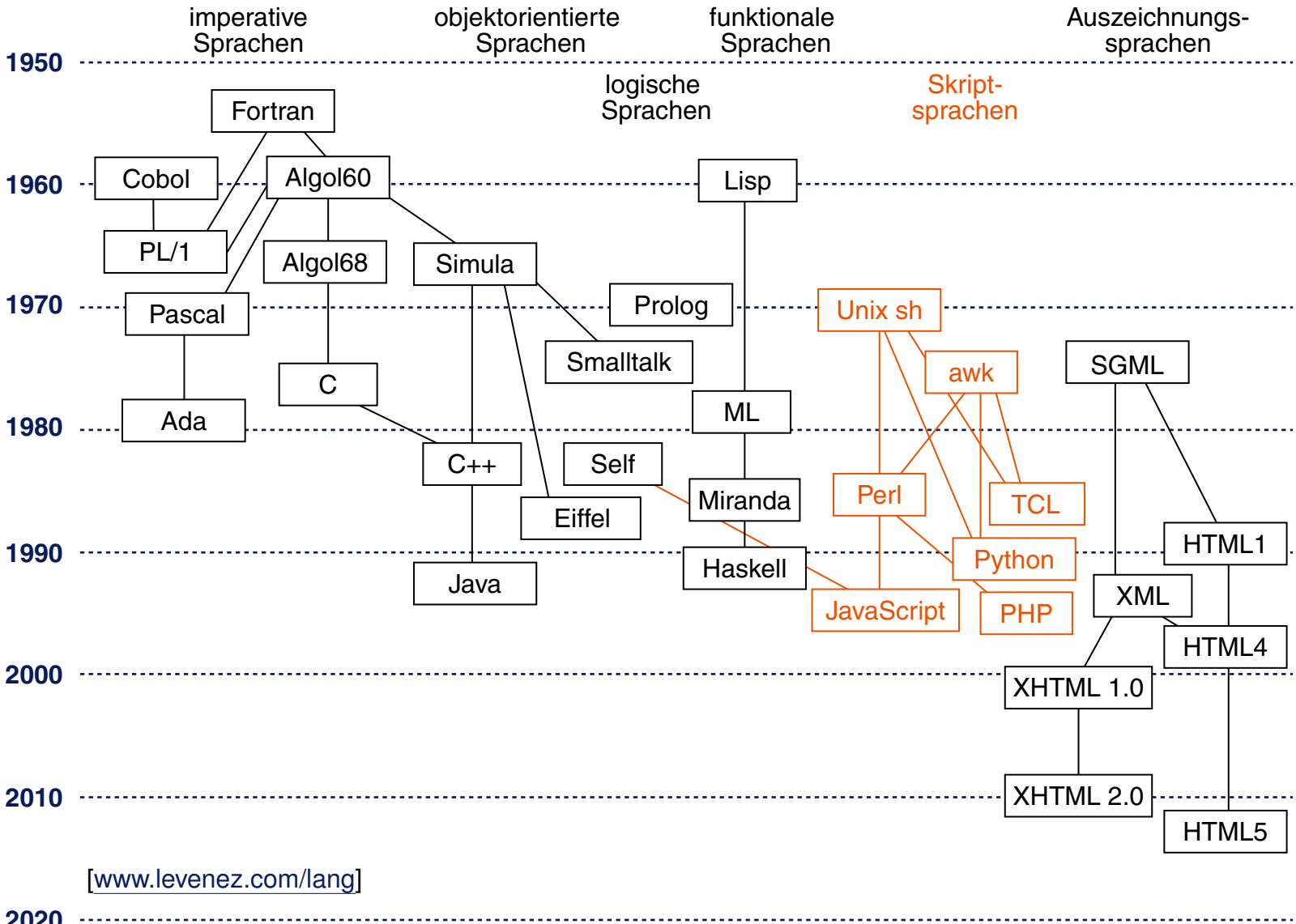


# Web-Client

Browser-Module [MDN [data flow](#)] [W3C [parsing model](#)] [[how browsers work](#)] [Google [1](#), [2](#), [3](#), [4](#)]



# Exkurs: Programmiersprachen





# Exkurs: Programmiersprachen [Kastens 2005]

## Ebenen von Spracheigenschaften

Ein Satz einer Sprache ist eine Folge von Zeichen eines gegebenen Alphabets.  
Zum Beispiel ist ein PHP-Programm ein Satz der Sprache PHP:

```
$line = fgets ( $fp , 64 ) ;
```

# Exkurs: Programmiersprachen [Kastens 2005]

## Ebenen von Spracheigenschaften

Ein Satz einer Sprache ist eine Folge von Zeichen eines gegebenen Alphabets.  
Zum Beispiel ist ein PHP-Programm ein Satz der Sprache PHP:

```
$line = fgets ( $fp , 64 ) ;
```

Die **Struktur** eines Satzes wird auf zwei Ebenen definiert:

1. Notation von Symbolen (Lexemen, Token).
2. Syntaktische Struktur.

Die **Bedeutung** eines Satzes wird auf zwei weiteren Ebenen an Hand der Struktur für jedes Sprachkonstrukt definiert:

3. Statische Semantik.  
Eigenschaften, die *vor* der Ausführung bestimmbar sind.
4. Dynamische Semantik.  
Eigenschaften, die *erst während* der Ausführung bestimmbar sind.

# Exkurs: Programmiersprachen [Kastens 2005]

## Ebene 1: Notation von Symbolen

Ein Symbol wird aus einer Folge von Zeichen des Alphabets gebildet. Die Regeln zur Notation von Symbolen werden durch **reguläre Ausdrücke** definiert.

```
$line = fgets ( $fp , 64 ) ;
```

# Exkurs: Programmiersprachen [Kastens 2005]

## Ebene 1: Notation von Symbolen

Ein Symbol wird aus einer Folge von Zeichen des Alphabets gebildet. Die Regeln zur Notation von Symbolen werden durch **reguläre Ausdrücke** definiert.

```
$line = fgets ($fp, 64);
```

Wichtige Symbolklassen in Programmiersprachen:

Symbolklasse	Beispiel in PHP
Bezeichner ( <i>Identifier</i> ) Verwendung: Namen für Variable, Funktionen, etc.	<code>\$line</code> , <code>fgets</code>
Literale ( <i>Literals</i> ) Verwendung: Zahlkonstanten, Zeichenkettenkonstanten	<code>64</code> , <code>"telefonbuch.txt"</code>
Wortsymbole ( <i>Keywords</i> ) Verwendung: kennzeichnen Sprachkonstrukte	<code>while</code> , <code>if</code>
Spezialzeichen Verwendung: Operatoren, Separatoren	<code>&lt;=</code> <code>=</code> <code>;</code> <code>{</code> <code>}</code>

## Bemerkungen:

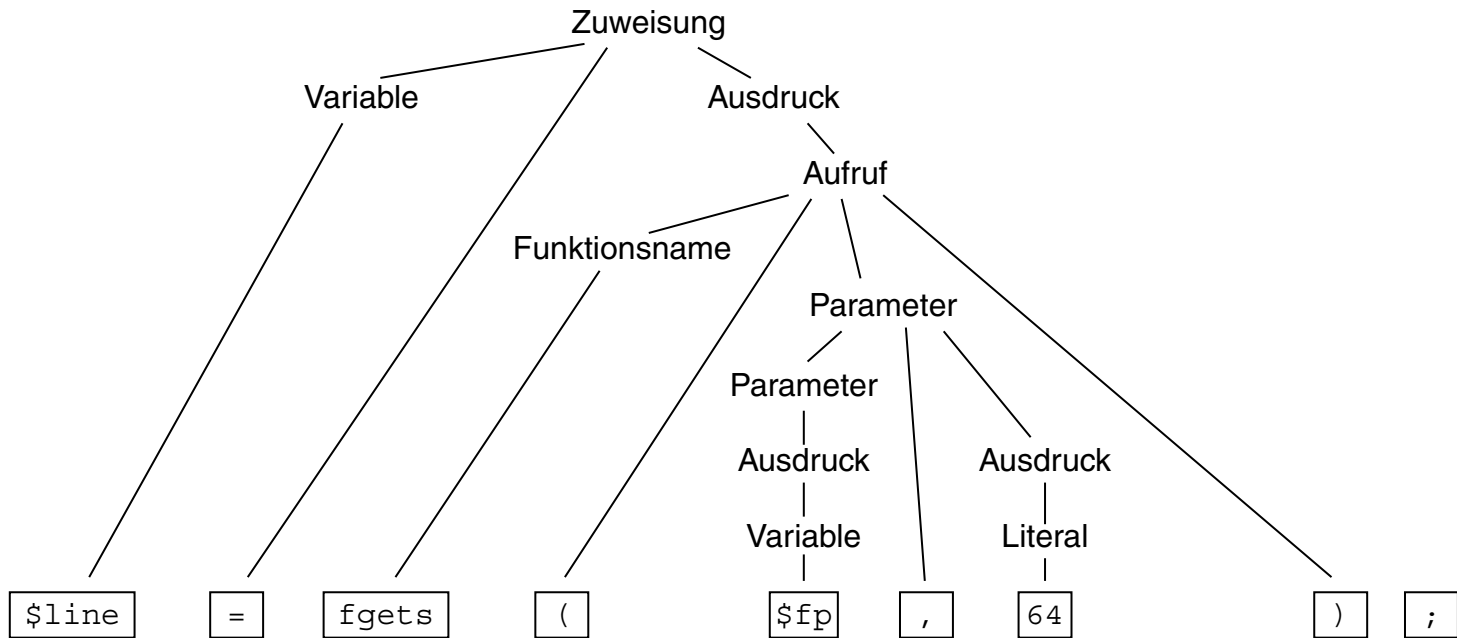
- ❑ Zwischenräume, Tabulatoren, Zeilenwechsel und Kommentare zwischen den Symbolen dienen der Lesbarkeit und sind sonst bedeutungslos.
- ❑ In Programmiersprachen bezeichnet der Begriff „Literal“ Zeichenfolgen, die zur Darstellung der Werte von Basistypen zulässig sind. Sie sind nicht benannt, werden aber über die jeweilige Umgebung ebenfalls in die Programmressourcen eingebunden. Literale können nur in rechtsseitigen Ausdrücken auftreten. Meist werden die Literale zu den Konstanten gerechnet und dann als literale Konstanten bezeichnet, da beide – im Gegensatz zu Variablen – zur Laufzeit unveränderlich sind.

Das Wort „Konstante“ im engeren Sinn bezieht sich allerdings mehr auf in ihrem Wert unveränderliche Bezeichner, d.h., eindeutig benannte Objekte, die im Quelltext beliebig oft verwendet werden können, statt immer das gleiche Literal anzugeben. [\[Wikipedia\]](#)

# Exkurs: Programmiersprachen [Kastens 2005]

## Ebene 2: Syntaktische Struktur

Ein Satz einer Sprache wird in seine Sprachkonstrukte gegliedert; sie sind meist ineinander geschachtelt. Diese syntaktische Struktur wird durch einen Strukturbaum dargestellt, wobei die Symbole durch Blätter repräsentiert sind:



Die Syntax einer Sprache wird durch eine **kontextfreie Grammatik** definiert. Die Symbole sind die Terminalsymbole der Grammatik.

## Ebene 3: Statische Semantik

Eigenschaften von Sprachkonstrukten, die ihre **Bedeutung** (Semantik) beschreiben, soweit sie anhand der Programmstruktur festgestellt werden können, ohne das Programm auszuführen (= statisch).

Elemente der statischen Semantik für übersetzte Sprachen:

- ❑ Bindung von Namen.

Regeln, die einer Anwendung eines Namens seine Definition zuordnen.

Beispiel: zu dem Funktionsnamen in einem Aufruf muss es eine Funktionsdefinition mit gleichem Namen geben.

## Ebene 3: Statische Semantik

Eigenschaften von Sprachkonstrukten, die ihre **Bedeutung** (Semantik) beschreiben, soweit sie anhand der Programmstruktur festgestellt werden können, ohne das Programm auszuführen (= statisch).

Elemente der statischen Semantik für übersetzte Sprachen:

- ❑ Bindung von Namen.

Regeln, die einer Anwendung eines Namens seine Definition zuordnen.

Beispiel: zu dem Funktionsnamen in einem Aufruf muss es eine Funktionsdefinition mit gleichem Namen geben.

- ❑ Typregeln.

Sprachkonstrukte wie Ausdrücke und Variablen liefern bei ihrer Auswertung einen Wert eines bestimmten Typs. Er muss im Kontext zulässig sein und kann die Bedeutung von Operationen näher bestimmen.

Beispiel: die Operanden des „\*“-Operators müssen Zahlwerte sein.



## Ebene 4: Dynamische Semantik

Eigenschaften von Sprachkonstrukten, die ihre Wirkung beschreiben und erst bei der Ausführung bestimmt oder geprüft werden können (= dynamisch).

Elemente der dynamischen Semantik:

- Regeln zur Analyse von Voraussetzungen, die für eine korrekte Ausführung eines Sprachkonstruktes erfüllt sein müssen.

Beispiel: ein numerischer Index einer Array-Indizierung, wie in `$var[$i]`, darf nicht kleiner als 0 sein.

# Exkurs: Programmiersprachen [Kastens 2005]

## Ebene 4: Dynamische Semantik

Eigenschaften von Sprachkonstrukten, die ihre Wirkung beschreiben und erst bei der Ausführung bestimmt oder geprüft werden können (= dynamisch).

Elemente der dynamischen Semantik:

- ❑ Regeln zur Analyse von Voraussetzungen, die für eine korrekte Ausführung eines Sprachkonstruktes erfüllt sein müssen.

Beispiel: ein numerischer Index einer Array-Indizierung, wie in `$var[$i]`, darf nicht kleiner als 0 sein.

- ❑ Regeln zur Umsetzung bestimmter Sprachkonstrukte.

Beispiel: Auswertung einer Zuweisung der Form

*Variable = Ausdruck*

Die Speicherstelle der Variablen auf der linken Seite wird bestimmt. Der Ausdruck auf der rechten Seite wird ausgewertet. Das Ergebnis ersetzt dann den Wert an der Stelle der Variablen. [[SELFHTML](#)]

## Bemerkungen:

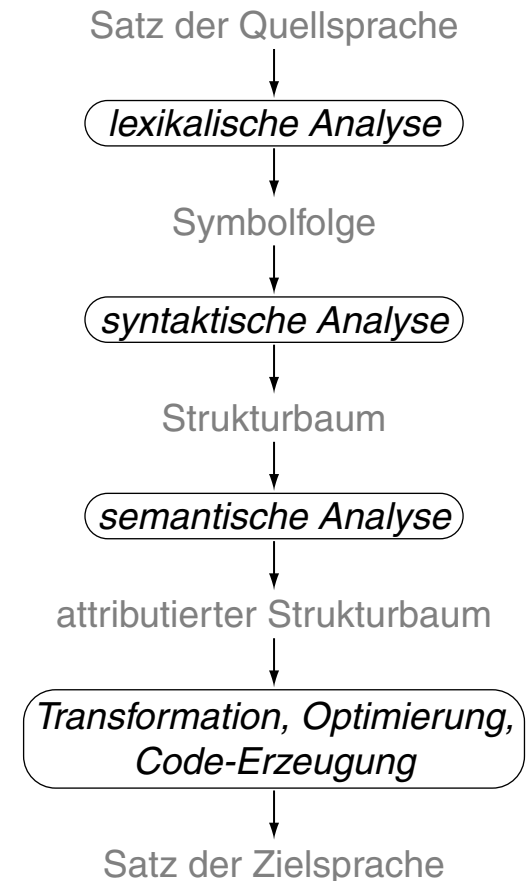
- ❑ Auf jeder der vier Ebenen gibt es also Regeln, die korrekte Sätze erfüllen müssen.
- ❑ In den Sprachen PHP und JavaScript gehören die Bindungsregeln zur statischen Semantik. Die Typregeln in diesen Sprachen gehören zur dynamischen Semantik, da sie erst bei der Ausführung des Programms anwendbar sind.

# Exkurs: Programmiersprachen [Kastens 2005]

## Übersetzung von Sprachen

Ein **Übersetzer** transformiert jeden korrekten Satz (Programm) der Quellsprache in einen gleichbedeutenden Satz (Programm) der Zielsprache.

- ❑ Die meisten Programmiersprachen zur Software-Entwicklung werden übersetzt. Beispiele: C, C++, Java, Ada, Modula.
- ❑ Zielsprache ist dabei meist eine Maschinensprache eines realen Prozessors oder einer abstrakten Maschine.
- ❑ Übersetzte Sprachen haben eine stark ausgeprägte statische Semantik.
- ❑ Der Übersetzer prüft die Regeln der statischen Semantik; viele Arten von Fehlern lassen sich vor der Ausführung finden.



# Exkurs: Programmiersprachen [Kastens 2005]

## Interpretation von Sprachen

Ein **Interpreter** liest einen Satz (Programm) einer Sprache und führt ihn aus.

Für Sprachen, die strikt interpretiert werden, gilt:

- ❑ sie haben eine einfache Struktur und keine statische Semantik
- ❑ Bindungs- und Typregeln werden erst bei der Ausführung geprüft
- ❑ nicht ausgeführte Programmteile bleiben ungeprüft

Beispiele: Prolog, interpretiertes Lisp

Moderne Interpreter erzeugen vor der Ausführung eine interne Repräsentation des Satzes; dann können auch Struktur und Regeln der statischen Semantik vor der Ausführung geprüft werden.

Beispiele: die Skriptsprachen JavaScript, PHP, Perl

## Bemerkungen:

- ❑ Es gibt auch Übersetzer für Sprachen, die keine einschlägigen Programmiersprachen sind: Sprachen zur Textformatierung ( $\text{\LaTeX}$  → PDF), Spezifikationssprachen (UML → Java).
- ❑ Interpretierer können auf jedem Rechner verfügbar gemacht werden und lassen sich in andere Software ([Web-Browser](#)) integrieren.
- ❑ Ein Interpretierer schafft die Möglichkeit einer weiteren Kapselung der Programmausführung gegenüber dem Betriebssystem.
- ❑ Interpretation kann 10-100 mal zeitaufwändiger sein, als die Ausführung von übersetztem Maschinencode.

# JavaScript

## Einführung [\[Einordnung\]](#)

### Charakteristika:

- ❑ interpretiert, dynamisch typisiert
- ❑ einfache objektorientierte Konzepte
- ❑ Notation ähnlich C++ und Java, konzeptuell wenig Bezug zu Java
- ❑ eng verknüpft mit HTML via DOM-API
- ❑ Interpretierer im [Web-Browser](#) integriert

# JavaScript

## Einführung [\[Einordnung\]](#)

### Charakteristika:

- ❑ interpretiert, dynamisch typisiert
- ❑ einfache objektorientierte Konzepte
- ❑ Notation ähnlich C++ und Java, konzeptuell wenig Bezug zu Java
- ❑ eng verknüpft mit HTML via DOM-API
- ❑ Interpretierer im [Web-Browser](#) integriert

### Anwendung:

- ❑ Programme, die im Web-Browser ausgeführt werden
- ❑ dynamischen Web-Seiten, Animationseffekte
- ❑ Reaktion auf Ereignisse bei der Interaktion mit Web-Seiten
- ❑ Programme, die Server-seitig ausgeführt werden



# JavaScript

## Einführung (Fortsetzung)

```
<!DOCTYPE html>
<html>

  <head>
    <meta http-equiv="content-type" content="text/html; ...">
    <title>Function</title>

    <script>
      function Quadrat() {
        let Zahl = document.QuadratForm.Eingabe.value;
        let Ergebnis = Zahl * Zahl;
        alert ("Das Quadrat von " + Zahl + " = " + Ergebnis);
      }
    </script>
  </head>

  <body>
    <form name="QuadratForm" id="QF" action="">
      <input type="text" name="Eingabe" size="3">
      <input type="button" value="Quadrat errechnen" onclick="Quadrat()">
    </form>
  </body>

</html>
```

# JavaScript

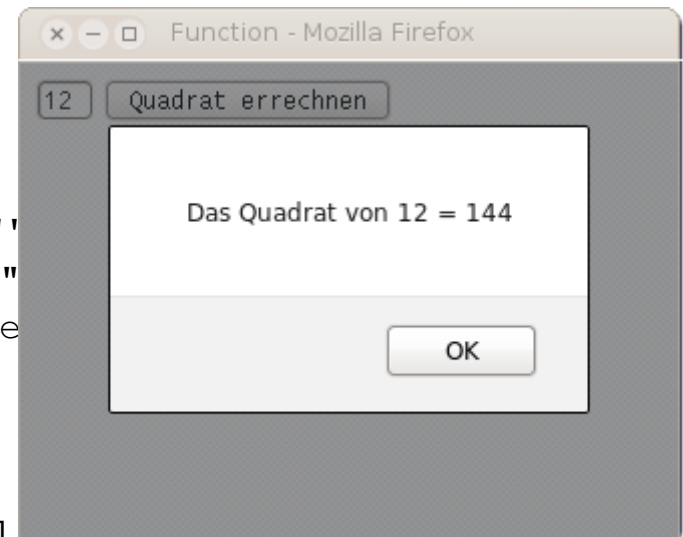
## Einführung (Fortsetzung)

```
<!DOCTYPE html>
<html>

  <head>
    <meta http-equiv="content-type" content="text/html; ...">
    <title>Function</title>

    <script>
      function Quadrat() {
        let Zahl = document.QuadratForm.Eingabe.value;
        let Ergebnis = Zahl * Zahl;
        alert ("Das Quadrat von " + Zahl + " = " + Ergebnis);
      }
    </script>
  </head>

  <body>
    <form name="QuadratForm" id="QF" action="">
      <input type="text" name="Eingabe" size="">
      <input type="button" value="Quadrat errechnen">
    </form>
  </body>
</html>
```



[\[JavaScript-Ausführung\]](#)

## Bemerkungen:

### ❑ JavaScript kompakt:

1. Historie
2. Einbindung in HTML-Dokumente
3. Grundlagen der Syntax
4. Variablen
5. Operatoren
6. Datentypen
7. Kontrollstrukturen
8. Funktionsbibliothek
9. Ereignisbehandlung

# JavaScript

## Historie

- 1993 NCSA Mosaic-Browser. Bilder im Fließtext, Farben für Links und Text.
- 1994 Netscape 1. Entwickelt von einer Splittergruppe des Mosaic-Teams.
- 1996 Netscape 2. Frames, JavaScript von [Brendan Eich](#). JavaScript heißt zunächst Mocha, dann LiveScript. Art des Dokumentzugriffs entspricht heutigem DOM Level 0.

# JavaScript

## Historie

- 1993 NCSA Mosaic-Browser. Bilder im Fließtext, Farben für Links und Text.
- 1994 Netscape 1. Entwickelt von einer Splittergruppe des Mosaic-Teams.
- 1996 Netscape 2. Frames, JavaScript von [Brendan Eich](#). JavaScript heißt zunächst Mocha, dann LiveScript. Art des Dokumentzugriffs entspricht heutigem DOM Level 0.
- 1996 Standardisierung der JavaScript Kernsprache durch die European Computer Manufacturers Association als **ECMAScript** in der Spezifikation [ECMA-262](#). [\[MDN\]](#)
- 1997 Netscape 4. „DHTML“, basierend auf W3C CSS 1 und JavaScript 1.2.
- 1997 Internet Explorer 4. Revolutionäres Konzept für dynamische Webseiten: W3C orientiert sich mit DOM-Entwicklung daran. Microsoft entwickelt JScript als Konkurrenz zu JavaScript.
- 1998 Netscape-Code wird Open Source, Mozilla-Projekt wird gestartet.

# JavaScript

## Historie

- 1993 NCSA Mosaic-Browser. Bilder im Fließtext, Farben für Links und Text.
- 1994 Netscape 1. Entwickelt von einer Splittergruppe des Mosaic-Teams.
- 1996 Netscape 2. Frames, JavaScript von [Brendan Eich](#). JavaScript heißt zunächst Mocha, dann LiveScript. Art des Dokumentzugriffs entspricht heutigem DOM Level 0.
- 1996 Standardisierung der JavaScript Kernsprache durch die European Computer Manufacturers Association als [ECMAScript](#) in der Spezifikation [ECMA-262](#). [\[MDN\]](#)
- 1997 Netscape 4. „DHTML“, basierend auf W3C CSS 1 und JavaScript 1.2.
- 1997 Internet Explorer 4. Revolutionäres Konzept für dynamische Webseiten: W3C orientiert sich mit DOM-Entwicklung daran. Microsoft entwickelt JScript als Konkurrenz zu JavaScript.
- 1998 Netscape-Code wird Open Source, Mozilla-Projekt wird gestartet.
- 2002 Mozilla Phoenix 0.1 (später Firefox). Open Source Rendering-Engine [Gecko](#).
- 2008 Google Chrome 1. Freie JavaScript-Engine V8 und JavaScript 1.7 ~ ECMAScript 3.
- 2010 „Letzte“ JavaScript-Version ist 1.8.5. Bezeichnung nun als ECMA-262 Editions. [\[MDN\]](#)
- 2012 ECMAScript 5. Übersicht über die Browser-Unterstützung: Alle modernen Browser unterstützen ECMAScript 5. [\[kangax\]](#)
- 2020 Statistiken zur Verbreitung: [\[tiobe.com\]](#) [\[redmonk.com\]](#)

## Bemerkungen:

- ❑ Die Entwicklung von JavaScript ist eng verknüpft mit dem „Browser-Krieg“ zwischen Microsoft und Netscape. Mehr zur JavaScript-Historie: [\[Tarquin\]](#) [\[SELFHTML\]](#)
- ❑ Ziel der W3C-DOM-Initiative war und ist es, die Browser-Entwicklung zu vereinheitlichen. Mittlerweile ermöglichen die Browser-APIs der verschiedenen Hersteller den Zugriff auf das HTML-Dokument gemäß der DOM Level 3 Spezifikation.
- ❑ Wiederholung: W3C DOM ist nicht nur für HTML-bezogene Skriptsprachen konzipiert, sondern bezieht sich auf alle Arten von Dokumenten, die in einer SGML-basierten Sprache geschrieben sind. [\[MDN\]](#) [\[SELFHTML\]](#)

# JavaScript

## Einbindung in HTML-Dokumente [[SELFHTML](#)]

1. Als Script-Bereich innerhalb eines HTML-Dokuments [JavaScript-Ausführung: [1](#), [2](#)]:

```
<script>  
...  
</script>
```



# JavaScript

## Einbindung in HTML-Dokumente [\[SELFHTML\]](#)

1. Als Script-Bereich innerhalb eines HTML-Dokuments [\[JavaScript-Ausführung: 1, 2\]](#) :

```
<script>  
...  
</script>
```

2. Innerhalb von HTML-Tags [\[JavaScript-Ausführung\]](#) :

Verwendung im Zusammenhang mit Ereignissen (*Events*), die ein Bediener auslösen kann.

- (a) Das **Ereignis ist als Attribut** codiert; der Attributwert ist eine Anweisungsfolge, die beim Eintritt des Ereignisses ausgeführt wird:

```
<input type="button" value="..." onclick="Quadrat()"> \[Beispiel\]
```

- (b) In einem Anker-Element kann – anstatt einer URL – mit `javascript:` eine Anweisungsfolge angegeben werden, die beim Klicken ausgeführt wird:

```
<a href="javascript:Quadrat()">...</a>
```

# JavaScript

## Einbindung in HTML-Dokumente [\[SELFHTML\]](#)

1. Als Script-Bereich innerhalb eines HTML-Dokuments [\[JavaScript-Ausführung: 1, 2\]](#) :

```
<script>  
...  
</script>
```

2. Innerhalb von HTML-Tags [\[JavaScript-Ausführung\]](#) :

Verwendung im Zusammenhang mit Ereignissen (*Events*), die ein Bediener auslösen kann.

- (a) Das **Ereignis ist als Attribut** codiert; der Attributwert ist eine Anweisungsfolge, die beim Eintritt des Ereignisses ausgeführt wird:

```
<input type="button" value="..." onclick="Quadrat()"> \[Beispiel\]
```

- (b) In einem Anker-Element kann – anstatt einer URL – mit `javascript:` eine Anweisungsfolge angegeben werden, die beim Klicken ausgeführt wird:

```
<a href="javascript:Quadrat()">...</a>
```

3. In einer separaten Datei [\[Source, JavaScript-Ausführung\]](#) :

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

## Bemerkungen:

- ❑ Das `<script>`-Element kann mehrfach in einem HTML-Dokument verwendet werden.
- ❑ Der JavaScript-Code eines Dokuments wird beim Einlesen des Dokuments vom Browser sofort ausgeführt.
- ❑ Die Auswertung einer Funktions*definition* erzeugt keine Ausgabe und liefert auch keinen Return-Wert.
- ❑ Es gibt keine Vorschrift dafür, an welcher Stelle in einem HTML-Dokument ein JavaScript-Bereich definiert werden darf. Aus Sicht der Ladezeit kann es sinnvoll sein, diesen Bereich am Ende eines HTML-Dokuments zu platzieren.
- ❑ Eine separate Datei mit JavaScript-Code sollte die Dateinamenerweiterung `.js` besitzen; insbesondere darf diese Datei nur JavaScript-Code enthalten. Das Encoding der Datei kann im einbindenden `<script>`-Element per `charset`-Attribut spezifiziert werden.

# JavaScript

## Grundlagen der Syntax [PHP]

Die Notation ähnelt in vieler Hinsicht der von C++ und Java.

### Bezeichner

- ❑ einheitliche Schreibweise für alle Arten von Bezeichnern:

`identifier = { letter | $ | _ } { letter | $ | _ | digit }*`

- ❑ Groß-/Kleinschreibung wird unterschieden (*case sensitive*)

# JavaScript

## Grundlagen der Syntax [PHP]

Die Notation ähnelt in vieler Hinsicht der von C++ und Java.

### Bezeichner

- ❑ einheitliche Schreibweise für alle Arten von Bezeichnern:

`identifizier = { letter | $ | _ } { letter | $ | _ | digit }*`

- ❑ Groß-/Kleinschreibung wird unterschieden (*case sensitive*)

### Anweisungen

- ❑ Ein Semikolon am Zeilenende ist möglich, kann aber entfallen.
- ❑ Zwischen Anweisungen in derselben Zeile muss ein Semikolon stehen.
- ❑ `//` kommentiert bis Zeilenende aus.
- ❑ Balancierte Kommentarklammerung: `/* Kommentar */`

# JavaScript

## Variablen und Konstanten [\[PHP\]](#) [\[MDN\]](#) [\[Wikipedia\]](#)

- ❑ Zur Deklaration von *Variablen* dienen die Schlüsselworte `var` und `let`.
- ❑ Zur Deklaration von *Konstanten* dient das Schlüsselwort `const`.
- ❑ Variablen und Konstanten können Werte beliebigen Typs annehmen.
- ❑ Unterscheidung von **lokalen** und **globalen** Variablen und Konstanten.
- ❑ Eine Variable ist lokal für eine Funktion, wenn sie innerhalb des Bindungsbereiches der Funktion mit `var` deklariert wird.
- ❑ Eine Variable (Konstante) ist lokal für einen Block, wenn sie innerhalb des Bindungsbereiches des Blocks mit `let` (`const`) deklariert wird.
- ❑ Globale Variablen und Konstanten gelten im ganzen Programm – es sei denn, sie werden von einer lokalen Variable oder Konstante überdeckt; lokale Variablen und Konstanten gelten nur in ihrem Bindungsbereich.
- ❑ Hoisting: unabhängig von ihrer Position gelten Deklarationen von Variablen und Funktionen im gesamten, zugehörigen Code-Kontext. [\[MDN\]](#)

# JavaScript

## Variablen und Konstanten [\[PHP\]](#) [\[MDN\]](#) [\[Wikipedia\]](#)

- ❑ Zur Deklaration von *Variablen* dienen die Schlüsselworte `var` und `let`.
- ❑ Zur Deklaration von *Konstanten* dient das Schlüsselwort `const`.
- ❑ Variablen und Konstanten können Werte beliebigen Typs annehmen.
- ❑ Unterscheidung von **lokalen** und **globalen** Variablen und Konstanten.
- ❑ Eine Variable ist lokal für eine Funktion, wenn sie innerhalb des Bindungsbereiches der Funktion mit `var` deklariert wird.
- ❑ Eine Variable (Konstante) ist lokal für einen Block, wenn sie innerhalb des Bindungsbereiches des Blocks mit `let` (`const`) deklariert wird.
- ❑ Globale Variablen und Konstanten gelten im ganzen Programm – es sei denn, sie werden von einer lokalen Variable oder Konstante überdeckt; lokale Variablen und Konstanten gelten nur in ihrem Bindungsbereich.
- ❑ Hoisting: unabhängig von ihrer Position gelten Deklarationen von Variablen und Funktionen im gesamten, zugehörigen Code-Kontext. [\[MDN\]](#)

# JavaScript

## Variablen: Illustration von Geltungsbereichen

```
let line, sum;

let col = 2;

let minimum = col,
    maximum = 999;

function compute(n)
{
    let sum = n;

    if (col > maximum) {
        var m1 = maximum;
        let m2 = m1;
        col = m2;
    }

    sum = sum * col;
    return sum;
}
```



# JavaScript

## Variablen: Illustration von Geltungsbereichen

```
let line, sum;

let col = 2;

let minimum = col,
    maximum = 999;

function compute(n) // n ist lokale Variable.
{
    let sum = n;      // sum ist lokale Variable.

    if (col > maximum) {
        var m1 = maximum; // m1 ist lokal für gesamte Funktion.
        let m2 = m1;
        col = m2;
    }

    sum = sum * col;    // col ist globale Variable.
    return sum;
}
```

# JavaScript

## Variablen: Illustration von Geltungsbereichen

```
let line, sum;

let col = 2;

let minimum = col,
    maximum = 999;

function compute(n) // n ist lokale Variable.
{
    let sum = n;      // sum ist lokale Variable.

    if (col > maximum) {
        var m1 = maximum; // m1 ist lokal für gesamte Funktion.
        let m2 = m1;      // m2 ist lokal für zugehörigen Block.
        col = m2;
    }

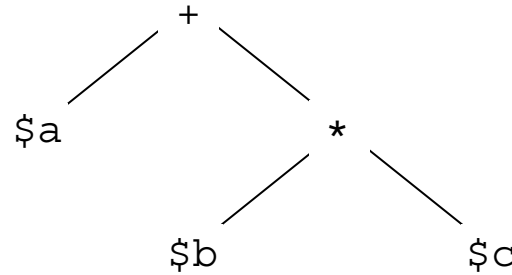
    sum = sum * col;    // col ist globale Variable.
    return sum;
}
```

# JavaScript

## Operatoren: Präzedenz, Assoziativität

Ein Operator mit höherer Präzedenz bindet seine Operanden stärker als ein Operator mit niedrigerer Präzedenz. Durch Klammerung lässt sich die Präzedenz in Termen vorschreiben. Beispiel:

$\$a + \$b * \$c$

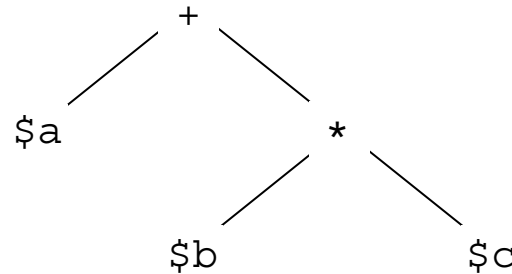


# JavaScript

## Operatoren: Präzedenz, Assoziativität

Ein Operator mit höherer Präzedenz bindet seine Operanden stärker als ein Operator mit niedrigerer Präzedenz. Durch Klammerung lässt sich die Präzedenz in Termen vorschreiben. Beispiel:

$\$a + \$b * \$c$



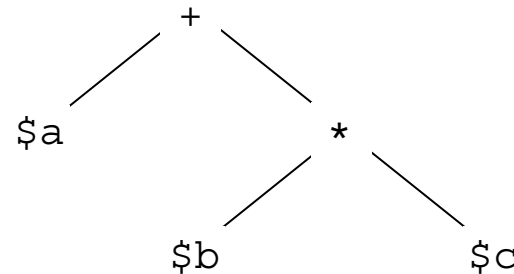
Ein Operator ist linksassoziativ (rechtsassoziativ), wenn beim Zusammentreffen von Operatoren **gleicher Präzedenz** der linke (rechte) Operator seine Operanden stärker bindet als der rechte (linke).

# JavaScript

## Operatoren: Präzedenz, Assoziativität

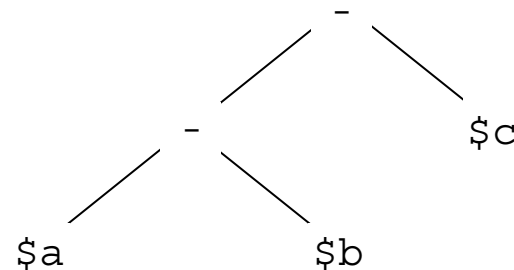
Ein Operator mit höherer Präzedenz bindet seine Operanden stärker als ein Operator mit niedrigerer Präzedenz. Durch Klammerung lässt sich die Präzedenz in Termen vorschreiben. Beispiel:

$\$a + \$b * \$c$



Ein Operator ist linksassoziativ (rechtsassoziativ), wenn beim Zusammentreffen von Operatoren **gleicher Präzedenz** der linke (rechte) Operator seine Operanden stärker bindet als der rechte (linke). Beispiel:

$\$a - \$b - \$c$



# JavaScript

## Operatoren: Übersicht [\[SELFHTML\]](#)

Präzedenz	Stelligkeit	Assoziativität	Operatoren	Erklärung
1	2	rechts	= += -=	Zuweisungsoperatoren
2	3	links	? :	bedingter Ausdruck
3	2	links		logische Disjunktion
4	2	links	&&	logische Konjunktion
5	2	links		Bitoperator
6	2	links	^	Bitoperator
7	2	links	&	Bitoperator
8	2	links	== != === !==	Gleichheit, Identität
9	2	links	< <= > >=	Ordnungsvergleich
10	2	links	<< >> >>>	shift-Operatoren
11	2	links	+ -	Konkatenation, Add., Subtr.
12	2	links	* / %	Arithmetik
13	1		! - ~	Negation (logisch, arithm.)
	1		++ --	Inkrement, Dekrement
	1		typeof void	Typabfragen, cast to undefined
14	1		() [] .	Aufruf, Index, Objektzugriff

# JavaScript

## Datentypen: Primitive [PHP]

### number

- ❑ Keine Unterscheidung zwischen Ganzzahlen und Gleitpunktzahlen.
- ❑ Der Wert `NaN` (*not a number*) steht für ein undefiniertes Ergebnis.
- ❑ Der Wert `Infinity` steht für einen Wert, der größer als die größte repräsentierbare Zahl ist.

### string

- ❑ Zeichenkettenlitterale mit einfachen oder doppelten Anführungszeichen.
- ❑ Konkatenation wie in Java: `let s = "Hello" + "world!"`
- ❑ Zeichenkettenfunktionen werden in objektorientierter Notation verwendet.  
Beispiele: `s.length`, `s.indexOf(substr)`, `s.charAt(i)`.

### boolean

- ❑ **Litterale:** `true` und `false` (insbesondere nicht: `True` bzw. `False`)
- ❑ **Operatoren:** Konjunktion `&&`, Disjunktion `||`, Negation `!`

# JavaScript

## Datentypen: Primitive (Fortsetzung)

`undefined`

- ❑ Der Wert `undefined` steht dafür, dass eine Variable keinen Wert hat.
- ❑ `undefined` wird zurückgegeben, falls (a) eine Variable benutzt wird, die zwar deklariert aber der nie ein Wert zugewiesen wurde oder (b) auf eine Objektkomponente zugegriffen wird, die nicht existiert.

`null`

- ❑ Der Wert `null` steht dafür, dass eine Variable keinen gültigen Wert hat.
- ❑ Obwohl `null` und `undefined` verschiedene Werte sind, werden sie vom Operator `==` als gleich interpretiert.
- ❑ Für typsichere Prüfung auf Gleichheit muss der Operator `===` verwendet werden bzw. `!==` für Ungleichheit.



# JavaScript

## Datentypen: Funktionen [\[MDN\]](#)

Eine Funktion ist ein Stück ausführbarer Code, der in einem JavaScript-Programm definiert ist. Funktionen sind Objekte vom Typ `Function`. Definitionsvarianten:

(a) Als „klassische“ Funktionsdeklaration:

(b) Als Funktionsausdruck bzw. Funktionsliteral [\[kangax\]](#):

(c) Mit dem Konstruktor `Function()`:

# JavaScript

## Datentypen: Funktionen [\[MDN\]](#)

Eine Funktion ist ein Stück ausführbarer Code, der in einem JavaScript-Programm definiert ist. Funktionen sind Objekte vom Typ `Function`. Definitionsvarianten:

- (a) Als „klassische“ **Funktionsdeklaration**:

```
function f(x, y) { return x*y };
```

- (b) Als Funktionsausdruck bzw. Funktionsliteral [\[kangax\]](#):

- (c) Mit dem Konstruktor `Function()`:

# JavaScript

## Datentypen: Funktionen [\[MDN\]](#)

Eine Funktion ist ein Stück ausführbarer Code, der in einem JavaScript-Programm definiert ist. Funktionen sind Objekte vom Typ `Function`. Definitionsvarianten:

(a) Als „klassische“ **Funktionsdeklaration**:

```
function f(x, y) { return x*y };
```

(b) Als **Funktionsausdruck** bzw. Funktionsliteral [\[kangax\]](#):

```
let p = (x, y) => { return x*y; };           // arrow (anonym)
```

oder

```
let q = function(x, y) { return x*y; };     // klassisch, anonym
```

oder

```
let r = function g(x, y) { return x*y; };   // klassisch, benannt
```

(c) Mit dem Konstruktor `Function()`:

# JavaScript

## Datentypen: Funktionen [\[MDN\]](#)

Eine Funktion ist ein Stück ausführbarer Code, der in einem JavaScript-Programm definiert ist. Funktionen sind Objekte vom Typ `Function`. Definitionsvarianten:

(a) Als „klassische“ **Funktionsdeklaration**:

```
function f(x, y) { return x*y };
```

(b) Als **Funktionsausdruck** bzw. Funktionsliteral [\[kangax\]](#):

```
let p = (x, y) => { return x*y; };           // arrow (anonym)
```

oder

```
let q = function(x, y) { return x*y; };     // klassisch, anonym
```

oder

```
let r = function g(x, y) { return x*y; };   // klassisch, benannt
```

(c) Mit dem Konstruktor `Function()`:

```
let s = new Function("x", "y", "return x*y;");
```

## Bemerkungen:

- ❑ Funktionen in JavaScript sind First-Class-Objekte und können gespeichert, als Parameter übergeben, zur Laufzeit eines Programms erstellt werden. [\[Wikipedia\]](#)
- ❑ Bei klassischen Funktionen wird das Schlüsselwort `this` abhängig vom Ort des Funktionsaufruf gebunden. Es zeigt auf den umgebenden Kontext (Aufruf als Funktion), das neue Objekt (Aufruf als Konstruktor mit `new`) oder das Ziel des Events (Aufruf als Event-Handler). [\[MDN\]](#)
- ❑ Besonderheiten zu Arrow-Funktionen [\[MDN\]](#):
  - Arrow-Funktionen können nicht als Konstruktor verwendet werden.
  - Besteht der Funktionskörper aus nur einer einzigen Anweisung, so ist die Verwendung der geschweiften Klammern optional und das `return`-Schlüsselwort implizit.
  - Enthält die Parameterliste nur einen Parameter, so ist die Verwendung der runden Klammern optional.

Beispiel: `f = x => x*x;      // f(2) ~> 4`

# JavaScript

## Datentypen: Objekte [vordefinierte Objekte]

Objekte bestehen aus Komponenten, die jeweils einen Namen und einen Wert haben. Objekte sind vom Typ `Object`. Definitionsvarianten:

(a) Mit einer Funktionsdeklaration als Konstruktor:

(b) Als Objektliteral [[w3schools](#)]:

(c) Mit der Funktion `Object ()` als Konstruktor:

# JavaScript

## Datentypen: Objekte [vordefinierte Objekte]

Objekte bestehen aus Komponenten, die jeweils einen Namen und einen Wert haben. Objekte sind vom Typ `Object`. Definitionsvarianten:

(a) Mit einer **Funktionsdeklaration** als Konstruktor:

```
function mycar(x) { this.year = x; return x*x };  
let car1 = new mycar(2020);  
car1.brand = "Tesla"
```

(b) Als Objektliteral [w3schools]:

(c) Mit der Funktion `Object ()` als Konstruktor:

# JavaScript

## Datentypen: Objekte [vordefinierte Objekte]

Objekte bestehen aus Komponenten, die jeweils einen Namen und einen Wert haben. Objekte sind vom Typ `Object`. Definitionsvarianten:

(a) Mit einer **Funktionsdeklaration** als Konstruktor:

```
function mycar(x) { this.year = x; return x*x };  
let car1 = new mycar(2020);  
car1.brand = "Tesla"
```

(b) Als **Objektliteral** [w3schools]:

```
let car2 = { year:2020, brand:"Tesla" };
```

(c) Mit der Funktion `Object()` als Konstruktor:



# JavaScript

## Datentypen: Objekte [vordefinierte Objekte]

Objekte bestehen aus Komponenten, die jeweils einen Namen und einen Wert haben. Objekte sind vom Typ `Object`. Definitionsvarianten:

(a) Mit einer **Funktionsdeklaration** als Konstruktor:

```
function mycar(x) { this.year = x; return x*x };  
let car1 = new mycar(2020);  
car1.brand = "Tesla"
```

(b) Als **Objektliteral** [w3schools]:

```
let car2 = { year:2020, brand:"Tesla" };
```

(c) Mit der Funktion `Object()` als Konstruktor:

```
let car3 = new Object();  
car3.year = 2020;  
car3.brand = "Tesla";
```

# JavaScript

## Datentypen: Objekte (Fortsetzung)

(d) Mit einem durch `class` definierten Konstruktor [\[MDN\]](#):

```
class MyCar {  
  constructor(x, y) {  
    this.year = x;  
    this.brand = y;  
  }  
}  
  
let car4 = new MyCar(2020, "Tesla");
```

# JavaScript

## Datentypen: Objekte (Fortsetzung)

(d) Mit einem durch `class` definierten Konstruktor [\[MDN\]](#):

```
class MyCar {  
  constructor(x, y) {  
    this.year = x;  
    this.brand = y;  
  }  
}  
  
let car4 = new MyCar(2020, "Tesla");
```

Zugriff auf Objektkomponenten mit *Objektausdruck.Komponente*:

`car1.year`  $\leadsto$  2020

...

`car4.year`  $\leadsto$  2020

## Bemerkungen:

- ❑ JavaScript ist keine objektorientierte Programmiersprache im Sinne von Java oder C++. Es gibt zwar den Datentyp `Object`, das Vererbungskonzept baut allerdings auf Objekt-Prototypen und nicht auf einem (abstrakten) Klassensystem auf.  
In JavaScript kann jedes Objekt als „Prototyp“ (zur Konstruktion neuer Objekte) verstanden werden, wodurch keine strikte Unterscheidung zwischen Klassen und Instanzobjekten existiert: Objekte lassen sich kopieren, Komponenten lassen sich hinzufügen und Vererbungshierarchien aufbauen. Stichwort: *prototypbasierte Vererbung* [\[MDN\]](#) [\[O'Reilly\]](#)
- ❑ In Konstruktoren und Methoden wird auf die Komponenten (Eigenschaften) mit dem Qualifier `this` zugegriffen; er bezeichnet die jeweilige mit `new` erzeugte Objektinstanz.
- ❑ *Objektausdruck* ist ein JavaScript-Ausdruck, der zu einer Referenz auf ein JavaScript-Objekt evaluiert.
- ❑ Objektkomponenten können Funktionen sein und heißen dann Methoden.
  - Aufruf von Methoden: *Objektausdruck.Komponente* ( )
  - Zugriff auf Wert der Eigenschaft: *Objektausdruck.Komponente*

# JavaScript

## Datentypen: Objekte (Fortsetzung)

Definition von Methoden als Funktionsausdruck oder Funktion:

```
class MyCircle {  
  constructor(r) {  
    this.radius = r;  
    this.circ = () => { return (Math.PI * this.radius * 2); };  
  }  
  
  area() {  
    return (Math.PI * this.radius * this.radius);  
  }  
}
```

# JavaScript

## Datentypen: Objekte (Fortsetzung)

Definition von Methoden als Funktionsausdruck oder Funktion:

```
class MyCircle {  
  constructor(r) {  
    this.radius = r;  
    this.circ = () => { return (Math.PI * this.radius * 2); };  
  }  
  
  area() {  
    return (Math.PI * this.radius * this.radius);  
  }  
}
```

Aufruf [\[JavaScript-Ausführung\]](#):

```
c = new MyCircle(3);  
document.writeln("Radius = " + c.radius);  
document.writeln("Area = " + c.area());  
document.writeln("Circumference = " + c.circ());
```

# JavaScript

Datentypen: Arrays [\[PHP\]](#) [\[JavaScript-Ausführung\]](#)

Ein Array ist eine Abbildung von Indizes auf Werte. Jedes Element eines Arrays ist ein Paar bestehend aus numerischem oder String-Index und zugeordnetem Wert. Arrays sind Objekte vom Typ `Array`.

Erzeugung von Arrays mit dem **Konstruktor** `Array()`:

- (a) Als Liste von Werten, indiziert von 0 an:
- (b) Durch Erweiterung eines leeren Arrays:
- (c) Als assoziatives Array:

# JavaScript

## Datentypen: Arrays [\[PHP\]](#) [\[JavaScript-Ausführung\]](#)

Ein Array ist eine Abbildung von Indizes auf Werte. Jedes Element eines Arrays ist ein Paar bestehend aus numerischem oder String-Index und zugeordnetem Wert. Arrays sind Objekte vom Typ `Array`.

Erzeugung von Arrays mit dem **Konstruktor** `Array()`:

(a) Als Liste von Werten, indiziert von 0 an:

```
let monatsName = new Array("", "Jan", ..., "Dez");
```

(b) Durch Erweiterung eines leeren Arrays:

```
let monatsName = new Array();  
monatsName[1] = "Jan"; monatsName[2] = "Feb"; ...
```

(c) Als assoziatives Array:

```
let monatsNr = new Array();  
monatsNr["Jan"] = 1; monatsNr["Feb"] = 2; ...
```



# JavaScript

## Datentypen: Arrays [\[PHP\]](#) [\[JavaScript-Ausführung\]](#)

Ein Array ist eine Abbildung von Indizes auf Werte. Jedes Element eines Arrays ist ein Paar bestehend aus numerischem oder String-Index und zugeordnetem Wert. Arrays sind Objekte vom Typ `Array`.

Erzeugung von Arrays mit dem **Konstruktor** `Array()`:

(a) Als Liste von Werten, indiziert von 0 an:

```
let monatsName = new Array("", "Jan", ..., "Dez");
```

(b) Durch Erweiterung eines leeren Arrays:

```
let monatsName = new Array();  
monatsName[1] = "Jan"; monatsName[2] = "Feb"; ...
```

(c) Als assoziatives Array:

```
let monatsNr = new Array();  
monatsNr["Jan"] = 1; monatsNr["Feb"] = 2; ...
```

Aufzählung aller Elemente **mit Schlüssel**:

```
for (let mname in monatsNr) {  
    document.writeln (mname + "->" + monatsNr[mname] + "<br/>");  
}
```

# JavaScript

## Kontrollstrukturen [\[PHP\]](#) [\[SELFHTML\]](#)

- ❑ Anweisungsfolge bzw. Block:
- ❑ Bedingte Anweisung:
- ❑ Return-Anweisung:
- ❑ `while`-Schleife:
- ❑ `for`-Schleife [\[JavaScript-Ausführung\]](#) :

# JavaScript

## Kontrollstrukturen [\[PHP\]](#) [\[SELFHTML\]](#)

### ❑ Anweisungsfolge bzw. Block:

```
{ let k = 42; document.writeln (5*k); }
```

Eine Anweisungsfolge definiert nur für `let`-Deklaration einen *Scope*. Eine `var`-Deklaration gilt auch in der umgebenden Funktion bzw. Programm.

### ❑ Bedingte Anweisung:

```
if (a < b) {min = a;} else {min = b;}
```

Bei einzelnen Anweisungen sind die `{}`-Klammern optional.

### ❑ Return-Anweisung:

```
return n*42;      return "*";
```

### ❑ while-Schleife:

### ❑ for-Schleife [\[JavaScript-Ausführung\]](#) :

# JavaScript

## Kontrollstrukturen [\[PHP\]](#) [\[SELFHTML\]](#)

### ❑ Anweisungsfolge bzw. Block:

```
{ let k = 42; document.writeln (5*k); }
```

Eine Anweisungsfolge definiert nur für `let`-Deklaration einen *Scope*. Eine `var`-Deklaration gilt auch in der umgebenden Funktion bzw. Programm.

### ❑ Bedingte Anweisung:

```
if (a < b) {min = a;} else {min = b;}
```

Bei einzelnen Anweisungen sind die `{}`-Klammern optional.

### ❑ Return-Anweisung:

```
return n*42;      return "*";
```

### ❑ while-Schleife:

```
i = 0; while (i < n) {document.write ("*"); ++i;}
```

### ❑ for-Schleife [\[JavaScript-Ausführung\]](#) :

```
for (let i = 0; i < n; ++i) {document.write ("*");}
```

# JavaScript

## Kontrollstrukturen [\[PHP\]](#) [\[SELFHTML\]](#)

### ❑ Anweisungsfolge bzw. Block:

```
{ let k = 42; document.writeln (5*k); }
```

Eine Anweisungsfolge definiert nur für `let`-Deklaration einen *Scope*. Eine `var`-Deklaration gilt auch in der umgebenden Funktion bzw. Programm.

### ❑ Bedingte Anweisung:

```
if (a < b) {min = a;} else {min = b;}
```

Bei einzelnen Anweisungen sind die `{}`-Klammern optional.

### ❑ Return-Anweisung:

```
return n*42;      return "*";
```

### ❑ while-Schleife:

```
i = 0; while (i < n) {document.write ("*"); ++i;}
```

### ❑ for-Schleife [\[JavaScript-Ausführung\]](#) :

```
for (let i = 0; i < n; ++i) {document.write ("*");}
```

### ❑ Parameterübergabe standardmäßig mittels **call-by-value**.

# JavaScript

## Funktionsbibliothek [\[PHP\]](#)

Ein Großteil der Funktionsbibliothek ist in Form von Objekten implementiert:

1. Vordefinierte Objekte. Funktionalität unabhängig von Dokument und Browser.

Beispiele: `Array`, `Date`, `Function`, `Math`, `Object`, `RegExp` [\[MDN\]](#)

### Vordefinierte Funktionen.

Beispiele: `eval`, `isFinite`, `isNaN`, `parseInt` [\[MDN\]](#)

# JavaScript

## Funktionsbibliothek [\[PHP\]](#)

Ein Großteil der Funktionsbibliothek ist in Form von Objekten implementiert:

1. Vordefinierte Objekte. Funktionalität unabhängig von Dokument und Browser.

Beispiele: `Array`, `Date`, `Function`, `Math`, `Object`, `RegExp` [\[MDN\]](#)

Vordefinierte Funktionen.

Beispiele: `eval`, `isFinite`, `isNaN`, `parseInt` [\[MDN\]](#)

2. DOM-Objekte. Repräsentation von Dokument und Browser. Beispiele:

Interface	DOM	DOM HTML
Node	<a href="#">[WHATWG]</a> <a href="#">[MDN]</a>	(Interface nicht erweitert)
Element, HTMLElement	<a href="#">[WHATWG]</a> <a href="#">[MDN]</a>	<a href="#">[W3C]</a> <a href="#">[WHATWG]</a> <a href="#">[MDN]</a>
Document	<a href="#">[WHATWG]</a> <a href="#">[MDN]</a>	<a href="#">[W3C]</a> <a href="#">[WHATWG]</a> <a href="#">[MDN]</a>
Window	(Interface nicht vorgesehen)	<a href="#">[W3C]</a> <a href="#">[WHATWG]</a> <a href="#">[MDN]</a>

## Bemerkungen:

- ❑ Die vordefinierten Objekte und Funktionen bilden den Kern der JavaScript-Sprache.  
Sprachreferenzen: [\[MDN\]](#) [\[w3schools\]](#)
- ❑ Vordefinierte Objekte werden auch als *Global Objects*, *Objects in the Global Scope* oder *Built-in Objects* bezeichnet.
- ❑ DOM-Objekte implementieren die Interfaces der sprachunabhängigen DOM-API. Dabei wird zwischen einem Kern-DOM ein HTML-DOM unterschieden. [\[MDN\]](#) [\[SELFHTML\]](#)

Die entsprechenden Interface-Referenzen:

- [\[W3C DOM HTML\]](#)
- [\[WHATWG DOM, DOM HTML\]](#) [\[MDN\]](#)



## Bemerkungen: (Fortsetzung)

- ❑ Die Wurzel der DOM-Objekt-Hierarchie ist das `Window`-Objekt. Eines der Kindobjekte ist das `Document`-Objekt; es bildet die Wurzel des HTML- bzw. XML-Dokuments.
- ❑ `Window` und `Document` sind die Objekte (Prototypen) gemäß der Interface-Spezifikation der DOM-API. In einem konkreten Dokument geschieht der Zugriff auf die entsprechenden Instanzen durch die JavaScript-Variablen `window` bzw. `document`; diese werden bei der Erzeugung eines neuen Browsing-Kontextes angelegt und initialisiert. [\[W3C\]](#) [\[WHATWG\]](#)
- ❑ Eines der Attribute von `Document` heißt `documentElement`; es verweist auf die Objektinstanz, die das Wurzelement (`<html>` oder `<xml>`) des HTML- bzw. XML-Dokuments repräsentiert. [\[WHATWG\]](#) [\[MDN\]](#)
- ❑ Illustration von Markup, DOM und gerenderter HTML-Seite im Live-DOM-Viewer. [\[hixie.ch\]](#)

# JavaScript

## Funktionsbibliothek: DOM-Objekte

Konzepte, um auf HTML-Elementobjekte und deren Eigenschaften zuzugreifen:

1. Qualifizierender Name gemäß der DOM-Hierarchie im Dokument.

Beispiel: `document.QuadratForm.Eingabe`

2. Methoden und Attribute der DOM-API. Beispiele:

`Dokumentausdruck.getElementsByName()`

`Dokumentausdruck.getElementById()`

`Dokumentausdruck.images`

`Dokumentausdruck.forms`

`{ Elementausdruck | Dokumentausdruck }.getElementsByTagName()`

`{ Elementausdruck | Dokumentausdruck }.querySelectorAll()`

3. Kombination von DOM-API und qualifizierendem Namen.

Beispiel: `document.getElementsByTagName("form")[0].Eingabe`

## Bemerkungen:

- ❑ Beim Parsen eines HTML-Dokuments durch den Browser (also mit dem Laden und Anzeigen eines Dokuments von einer URL) wird das DOM gemäß der Spezifikation der DOM-API instanziiert: neben den Instanzen des `Window`- und `Document`-Objekts wird für jedes HTML-Element eine entsprechende Objektinstanz erzeugt und verlinkt. Damit stehen alle für ein HTML-Element erlaubten Attribute als Objekteigenschaften im DOM zur Verfügung. Diese Datenstruktur bildet die Dokumentrepräsentation im Browser.
- ❑ *Elementausdruck* und *Dokumentausrdruck* sind Spezialisierungen von *Objektausdruck* gemäß der Semantik der DOM-Hierarchie und evaluieren zu einer Referenz auf ein entsprechendes JavaScript-Objekt.
- ❑ *Objektausdruck* notiert einen Pfad in einer Objekthierarchie und kann Objekte, Methoden und Variablen kombinieren; diese können gemäß der DOM-API als auch benutzerdefiniert sein.  
Beispiel:

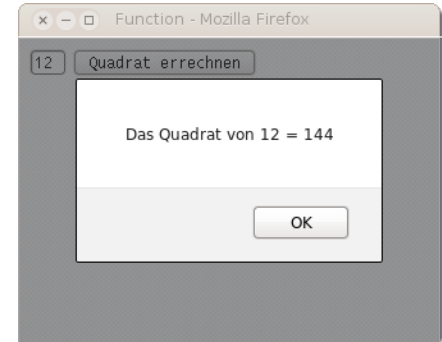
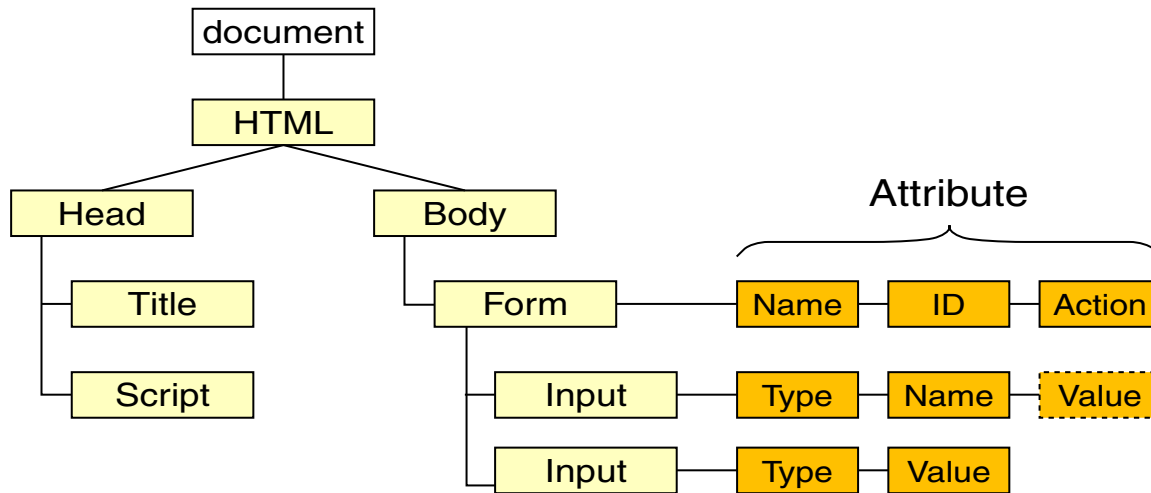
```
document .getElementsByTagName ("form") [0] .Eingabe .value
```

The diagram illustrates the components of the JavaScript expression `document .getElementsByTagName ("form") [0] .Eingabe .value` based on the DOM API hierarchy:

- `document`: DOM-API-Variable
- `.getElementsByTagName ("form")`: DOM-API-Methode
- `[0]`: benutzerdefiniertes Objekt
- `.Eingabe`: benutzerdefiniertes Objekt
- `.value`: DOM-API-Objekteigenschaft

# JavaScript

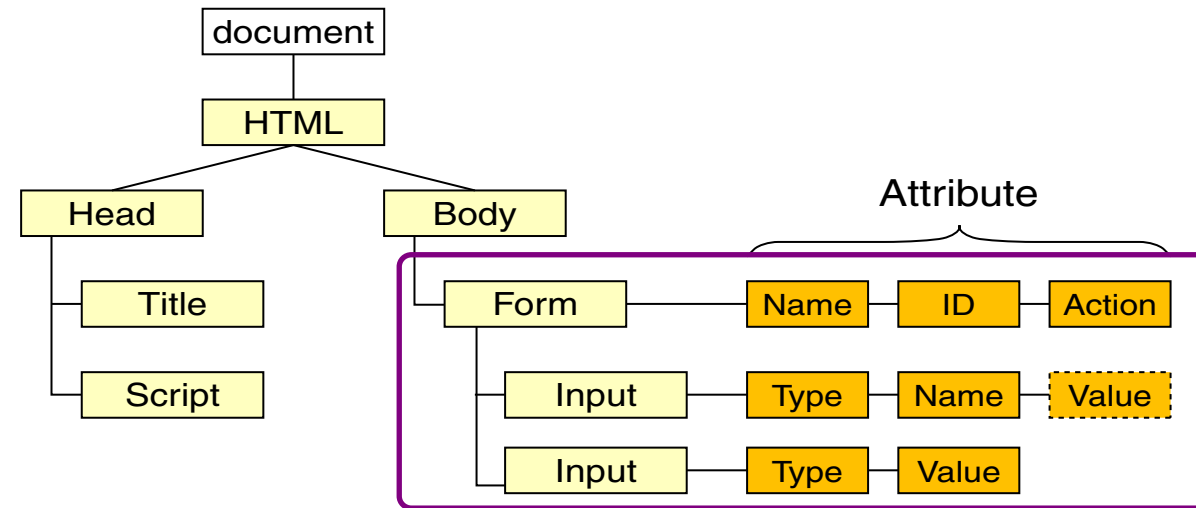
## Funktionsbibliothek: DOM-Objekte (Fortsetzung) [JavaScript-Einführungsbeispiel]



```
<!DOCTYPE html>
<html>
  <head>
    ...
    <script>
      function Quadrat() {...}
    </script>
  </head>
  <body>
    <form name="QuadratForm" id="QF" action="">
      <input type="text" name="Eingabe" size="3">
      <input type="button" value="Quadrat errechnen" onclick="Quadrat()">
    </form>
  </body>
</html>
```

# JavaScript

## Funktionsbibliothek: DOM-Objekte (Fortsetzung) [[JavaScript-Einführungsbeispiel](#)]



### Zugriffsmöglichkeiten auf das erste **Formelement**:

```
document.QuadratForm
document.getElementsByTagName("form")[0]
document.getElementById("QF")
document.querySelector("body #QF")
```

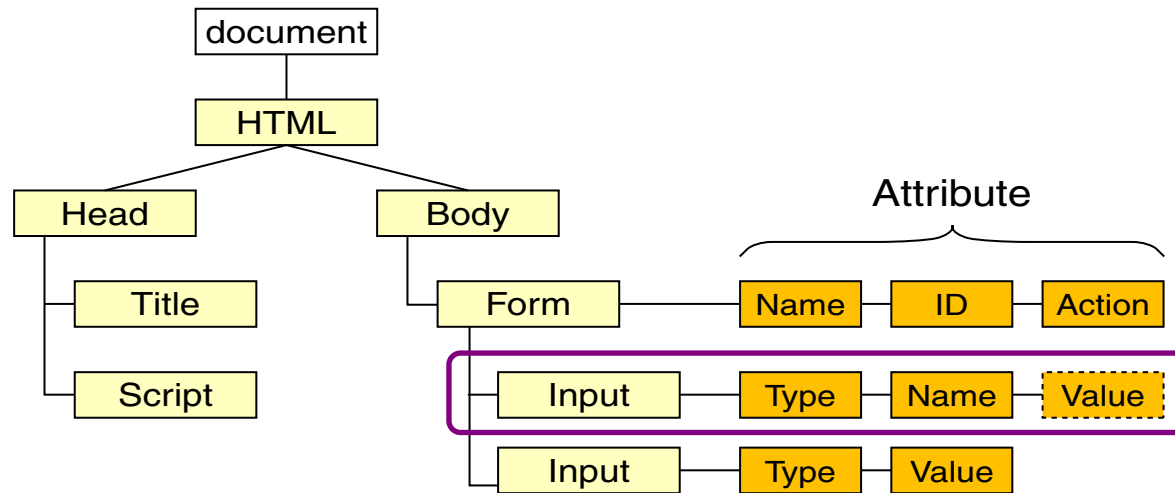
qualifizierender Name  
DOM-API  
DOM-API  
DOM-API

### Kontrollausgaben:

```
document.writeln(document.QuadratForm) ~> [object HTMLFormElement]
document.writeln(document.getElementsByTagName("form")) ~> [object HTMLCollection]
document.writeln(document.getElementsByTagName("form")[0]) ~> [object HTMLFormElement]
```

# JavaScript

## Funktionsbibliothek: DOM-Objekte (Fortsetzung) [[JavaScript-Einführungsbeispiel](#)]



### Zugriffsmöglichkeiten auf das erste Eingabeelement:

`document.QuadratForm.Eingabe`

`document.getElementsByName("Eingabe")[0]`

`document.getElementsByTagName("form")[0].Eingabe`

`document.querySelector("body #QF").Eingabe`

qualifizierender Name

DOM-API

Kombination

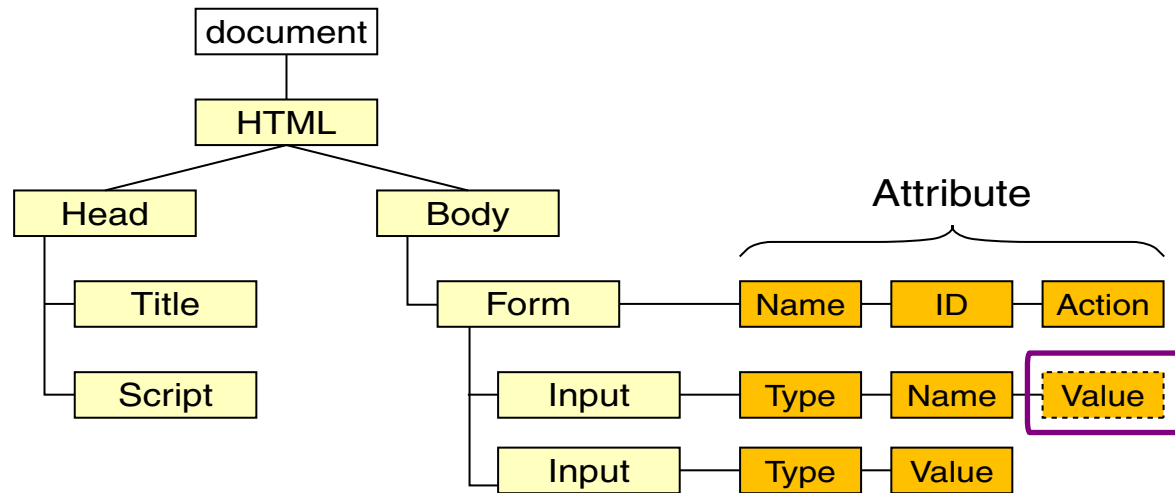
Kombination

### Kontrollausgaben:

`document.writeln(document.QuadratForm.Eingabe) ~> [object HTMLInputElement]`

# JavaScript

## Funktionsbibliothek: DOM-Objekte (Fortsetzung) [JavaScript-Einführungsbeispiel]



Zugriffsmöglichkeiten auf das **Eingabefeld** im ersten Eingabeelement:

```
document.QuadratForm.Eingabe.value  
document.getElementsByName("Eingabe")[0].value  
document.getElementsByTagName("form")[0].Eingabe.value  
document.querySelector("body #QF").Eingabe.value
```

Kontrollausgaben:

```
document.writeln(document.QuadratForm.Eingabe.value) ~ 11
```

# JavaScript

## Funktionsbibliothek: DOM-Objekte (Fortsetzung) [[JavaScript-Einführungsbeispiel](#)]

```
<script>
  function Quadrat() {
    let Zahl = document.QuadratForm.Eingabe.value;
    let Ergebnis = Zahl * Zahl;
    alert ("Das Quadrat von " + Zahl + " = " + Ergebnis);
  }
</script>

<form name="QuadratForm" id="QF" action="">
  <input type="text" name="Eingabe" size="3">
  <input type="button" value="Quadrat errechnen" onclick="Quadrat()">
</form>
```

Genauso wie das Abfragen ist auch das Setzen von Werten möglich:

```
document.QuadratForm.Eingabe.value = 12;
```



# JavaScript

## Ereignisbehandlung

Ein Ereignis (*Event*) ist die Wahrnehmung einer Zustandsänderung. Die ereignisgetriebene Programmierung ordnet den Ereignissen Operationen zu.

# JavaScript

## Ereignisbehandlung

Ein Ereignis (*Event*) ist die Wahrnehmung einer Zustandsänderung. Die ereignisgetriebene Programmierung ordnet den Ereignissen Operationen zu.

Varianten für die Behandlung des Ereignisses „Mausklick“:

```
<!DOCTYPE html>
<html>
  <head><title>Event</title></head>
  <body>
    <form name="testForm">
      <input type="button" value="ping" onclick="alert('ping!')">
      <input type="button" value="pong" name="Knopf">
    </form>

    <script>
      document.testForm.Knopf.onclick = function(){ alert("pong!") };
      document.testForm.Knopf.addEventListener("click",
                                                function(){ alert("pong!") });
    </script>
  </body>
</html>
```

# JavaScript

## Ereignisbehandlung

Ein Ereignis (*Event*) ist die Wahrnehmung einer Zustandsänderung. Die ereignisgetriebene Programmierung ordnet den Ereignissen Operationen zu.

Varianten für die Behandlung des Ereignisses „Mausklick“:

```
<!DOCTYPE html>
<html>
  <head><title>Event</title></head>
  <body>
    <form name="testForm">
      <input type="button" value="ping" onclick="alert('ping!')">
      <input type="button" value="pong" name="Knopf">
    </form>

    <script>
      document.testForm.Knopf.onclick = function(){ alert("pong!"); };
      document.testForm.Knopf.addEventListener("click",
                                                function(){ alert("pong!"); });
    </script>
  </body>
</html>
```

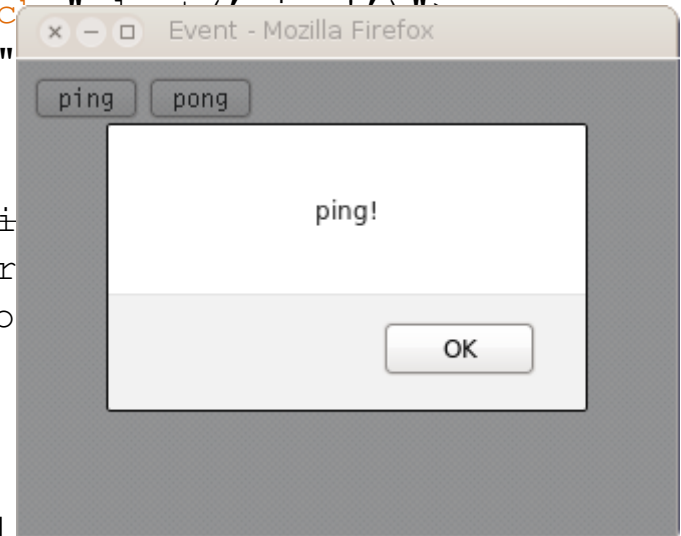
# JavaScript

## Ereignisbehandlung

Ein Ereignis (*Event*) ist die Wahrnehmung einer Zustandsänderung. Die ereignisgetriebene Programmierung ordnet den Ereignissen Operationen zu.

Varianten für die Behandlung des Ereignisses „Mausklick“:

```
<!DOCTYPE html>
<html>
  <head><title>Event</title></head>
  <body>
    <form name="testForm">
      <input type="button" value="ping" onclick="ping()" />
      <input type="button" value="pong" name="Knopf" />
    </form>
    <script>
      document.testForm.Knopf.onclick = function() {
        document.testForm.Knopf.addEventListener('click', function() {
          alert('ping!');
        });
      };
    </script>
  </body>
</html>
```



[JavaScript-Ausführung]

## Bemerkungen:

- ❑ Beachte, dass Funktionen über Attribute im HTML-Markup (Zeile 6) oder direkt via DOM-Interface (Zeile 10) zugewiesen werden können.
- ❑ Bei der Zuweisung via DOM-Interface kann das entsprechende Event-Attribut als Teil der DOM-Hierarchie (Zeile 10) oder die API-Funktion `addEventListener` (Zeile 11) verwendet werden. Die letzte Variante ist zu bevorzugen, da sich hiermit einem Element für das gleiche Ereignis mehrere Handler zuweisen lassen. [\[WHATWG\]](#) [\[MDN\]](#)
- ❑ JavaScript kennt über 70 verschiedene Ereignisse. [\[MDN\]](#)
- ❑ Mittlerweile können viele Maus-Events ohne JavaScript mittels CSS realisiert werden. Beispiel: [Webis: [Courses Map](#)]

# JavaScript

## Quellen zum Nachlernen und Nachschlagen im Web

- ❑ ECMA. *Standard ECMA-262: ECMAScript Language Specification*.  
[www.ecma-international.org/publications/standards/Ecma-262](http://www.ecma-international.org/publications/standards/Ecma-262)
- ❑ Kastens. *Einführung in Web-bezogene Sprachen*.  
Vorlesung WS 2005/06, Universität Paderborn.
- ❑ MDN. *JavaScript*.  
[developer.mozilla.org/en-US/docs/Web/JavaScript](http://developer.mozilla.org/en-US/docs/Web/JavaScript)
- ❑ O'Reilly. *JavaScript. The Definitive Guide*  
[docstore.mik.ua/oreilly/webprog/jscript](http://docstore.mik.ua/oreilly/webprog/jscript)
- ❑ SELFHTML e.V. *JavaScript*.  
[wiki.selfhtml.org/wiki/JavaScript](http://wiki.selfhtml.org/wiki/JavaScript)
- ❑ W3 Schools. *JavaScript Tutorial*.  
[www.w3schools.com/js](http://www.w3schools.com/js)
- ❑ Wenz. *JavaScript und AJAX*.  
[openbook.rheinwerk-verlag.de/javascript\\_ajax](http://openbook.rheinwerk-verlag.de/javascript_ajax)