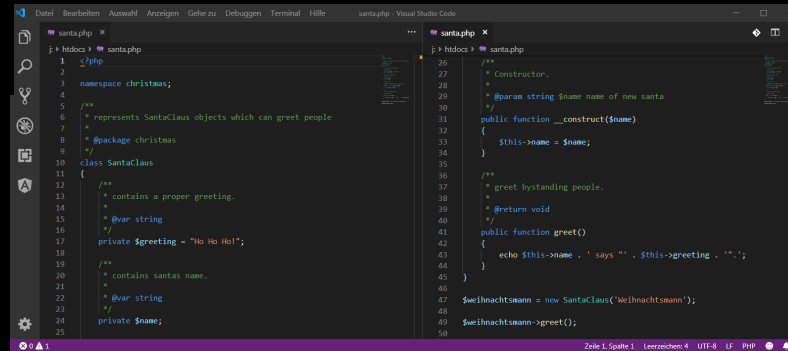


# Crashkurs Webprogrammierung



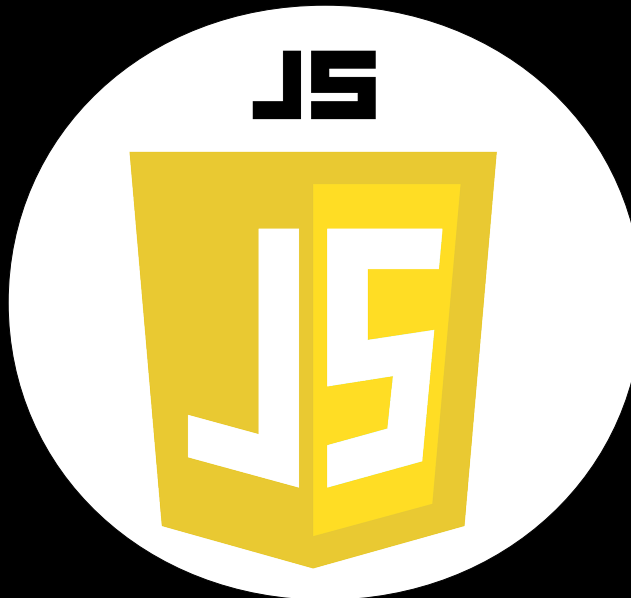
The screenshot shows two editor windows in Visual Studio Code. The left window displays a PHP file named 'santap.php' with the following code:

```
1 <?php
2
3 namespace christmas;
4
5 /**
6  * represents SantaClaus objects which can greet people
7  */
8 * @package christmas
9 */
10 class SantaClaus
11 {
12     /**
13      * contains a proper greeting.
14      */
15     * @var string
16     */
17     private $greeting = "Ho Ho Ho!";
18
19     /**
20      * contains santas name.
21      */
22     * @var string
23     */
24     private $name;
25 }
```

The right window displays the same file with the following code:

```
26 /**
27  * Constructor.
28  */
29 * @param string $name name of new santa
30 */
31 public function __construct($name)
32 {
33     $this->name = $name;
34 }
35
36 /**
37  * greet bystanding people.
38  */
39 * @return void
40 */
41 public function greet()
42 {
43     echo $this->name . ' says ' . $this->greeting . ' ' . ' ';
44 }
45
46 $sachnachtsmann = new SantaClaus("Weihnachtsmann");
47 $sachnachtsmann->greet();
48
49
```

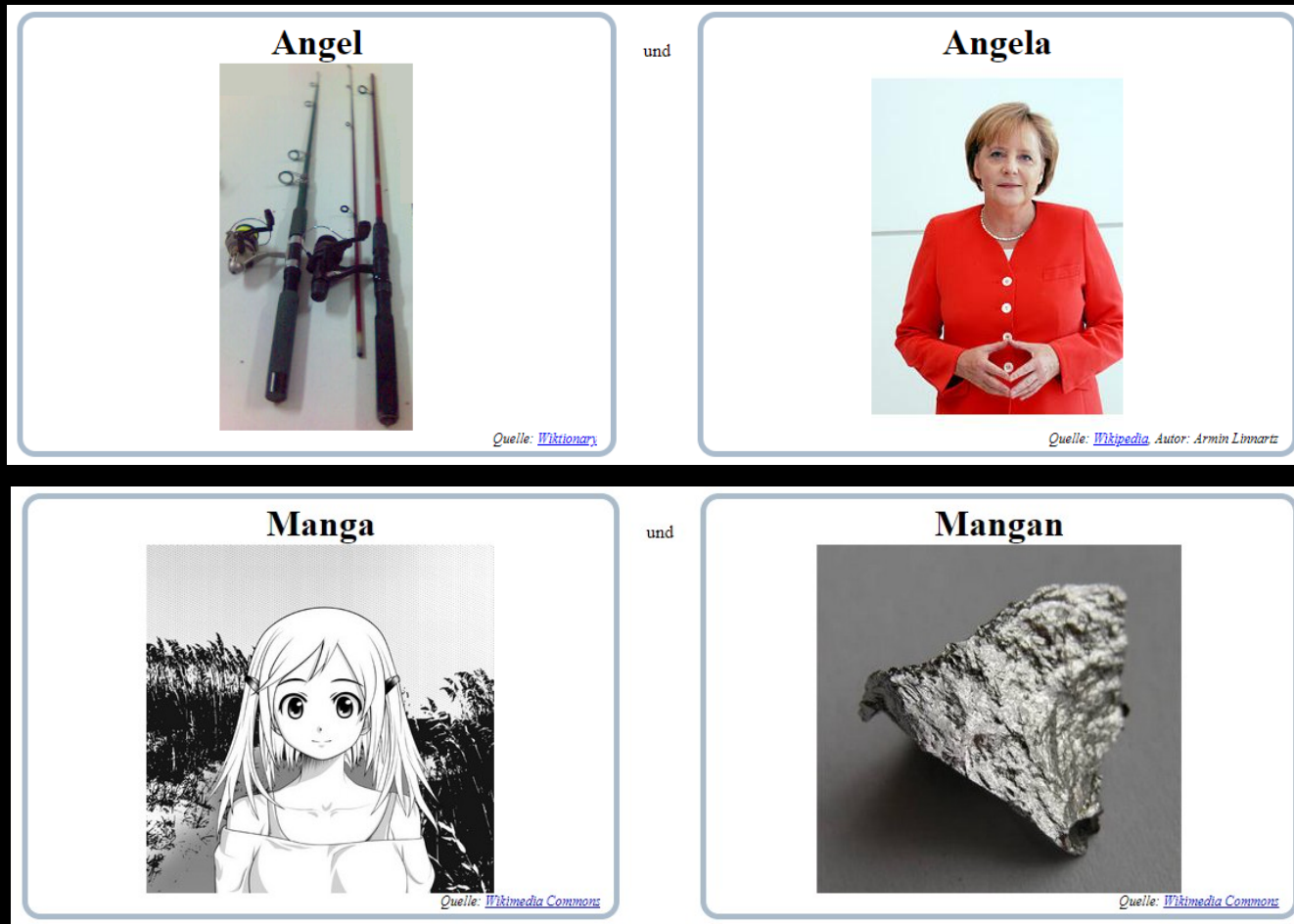
## Clientseitige Programmierung mit JavaScript



# Was hat JavaScript mit Java zu tun?



# Der Unterschied zwischen Java und JavaScript ist größer als der zwischen...



(C) Matthias Apsel, <http://selfhtml.apsel-mv.de/java-javascript>

# JavaScript ist...

- ...eine Skriptsprache (Programmiersprache), die im Browser interpretiert und ausgeführt wird
- ...*ECMAScript* mit *Document Object Model (DOM)*
- ...eine Sprache für prozedurale, objektorientierte oder funktionale Programmierung ohne statische Typisierung
- ...für interaktive Elemente auf Webseiten zuständig
- ...nicht immer und überall vollständig verfügbar  
(Erinnerung: *progressive enhancement* wo möglich!)
- ...in der Syntax mit Java und C verwandt, verhält sich manchmal aber ganz anders

# Was ist prozedurale Programmierung?



# Prozedurale Programmierung

Prozedural programmierte Skripte werden Anweisung für Anweisung von oben nach unten ausgeführt. Teile des Programms sind in Funktionen gekapselt, die im Programmablauf beliebig aufgerufen werden können.

```
function addiere(a, b) {  
    let summe = a + b;  
    // einzeliger Kommentar  
    return summe;  
}
```

```
let x = 2.5;           // mit let werden Variablen definiert  
var y = x*2;          // mit var auch, dann aber mit ungewöhnlichem Scope
```

```
/* Kommentar, der über mehrere Zeilen gehen kann */  
const ergebnis = addiere(x,y); // Konstanten sind unveränderbar  
  
console.log("Die Summe aus "+x+" und "+y+" ist: "+ergebnis+".");
```

Ausprobieren mit:  
Browser → F12 → Konsole



# Verschiedene Werttypen

Eine JavaScript-Variable kann einen Wert eines beliebigen Typs aufnehmen. Umwandlungen (für Berechnungen etc.) finden automatisch statt.

```
/* Primitive Typen */
var bool = true;           // Boolean: Wahrheitswert, true oder false
var zahl = 15.5;           // Number: Ganzzahl oder Dezimalzahl
var str = "blabla";        // String: Zeichenkette (in "" oder '')

/* Arrays, Objekte und Funktionen */
var arr = [1, 2, 3, 4];    // Array: Liste von Werten; Index ab 0
arr[2] = "drei";           // vorhandener Index: Wert verändern
arr[4] = 5;                // nicht vorhandener Index: Wert hinzufügen
var obj = {                // Objekt mit diversen Eigenschafts-Variablen
  text: "blabla",          // Zugriff durch obj.text oder obj["text"]
  zahl: 1.5                // das Komma ganz hinten kann man weggelassen
}
var fun = function(a, b) { /*...*/ } // Funktion mal anders definiert
```

In den Standardobjekten gibt es viele sinnvolle Funktionen zum Umgang mit Variablen und ihren Werten → <https://wiki.selfhtml.org/wiki/JavaScript/Objekte>

Testen! Dann: bool + zahl



# Kontrollstrukturen: Schleifen

Mit Schleifen werden Programmteile öfters ausgeführt.

```
var zahlen = [1,2,3,4,5], i = 10;
// Die for-Schleife, sinnvoll z.B. zum Laufen über Array oder Strings
for (var i = 0; i < zahlen.length; i++) {    // init; Bedingung; incr;
    console.log("Index "+i+": "+zahlen[i]);    // i++ bedeutet i = i+1;
}
console.log(i);
// Die while-Schleife, wiederholt solange Bedingung erfüllt ist
while (zahlen.length > 2) {                // while(false) wäre Endlosschleife!
    console.log(zahlen.pop());            // pop wirft letztes Element raus
}
// Die for-of-Schleife, nur für iterierbare Typen (String, Array,...)
for (var wert of zahlen) {                // Achtung: IE11 kann das nicht!
    console.log(wert);                    // im Zweifel for-Schleife benutzen
}
var objekt = { z: zahlen, x: 2 };
// Die for-in-Schleife, um Objekt-Eigenschaften zu durchlaufen
for (var eigenschaft in objekt) {        // das geht auch mit Arrays etc.
    console.log(eigenschaft + ": " + objekt[eigenschaft]);
}
```

Ausgabe überlegen und prüfen! Was ändert sich mit *let* statt *var*?





# Kontrollstrukturen: Verzweigungen

Anweisungen nur unter bestimmten Umständen ausführen

```
var x = 3, text = '';

if (x == 2) {           // Achtung: zwei ==-Zeichen für Vergleich
    console.log("x ist 2"); // andere Vergleich-Zeichen: !=, <=, ...
} else {
    console.log("x ist nicht 2, sondern " + x);
}

switch(x) {             // switch nimmt Fallunterscheidung vor
    case 1:
        text = 'x ist sehr klein.';
        break;           // ohne break werden nachfolgende Fälle
    case 2:               // mit ausgeführt! Außerdem: Ein break
    case 3:               // in einer Schleife beendet sie sofort
        text = 'x ist nicht rießig.';
        break;
    default:              // default wird ausgeführt, wenn sonst
        text = 'was soll denn das sein?'; // nichts passt
        break;
}

console.log(text);
```

Mit verschiedenen x-Werten testen!



# Was ist objektorientierte Programmierung?



# OOP mit JavaScript

## Funktionen können als Konstruktor dienen!

```
function Auto(farbe, marke) {
  this.farbe = farbe;
  this.marke = marke;
  this.hupen = function() {
    // Kurzschreibweise: Bedingung?Ja-Rückgabe:Nein-Rückgabe
    let marke = (this.marke === undefined)?"Wagen":this.marke;
    console.log("Der "+this.farbe+"e "+marke+" hupt!");
  }
}

let a = new Auto("blau");           // Aufruf mit nur einem Argument
let b = new Auto("rot", "Porsche"); // Aufruf mit zwei Argumenten
a.hupen();
b.hupen();
```

Testen!



Das ist die einfachste Möglichkeit, eine Klasse zu realisieren. Wird Vererbung benötigt, so muss die Klasse über ihren Prototyp realisiert und vererbt werden. Siehe z.B.: [https://wiki.selfhtml.org/wiki/JavaScript/Vererbung#Prototypische\\_Objekte\\_.28Prototypen.29](https://wiki.selfhtml.org/wiki/JavaScript/Vererbung#Prototypische_Objekte_.28Prototypen.29)

# Und wie können wir das jetzt im Browser anwenden?



# JavaScript in HTML einbinden

- JavaScript kann direkt im HTML innerhalb eines `<script>`-Elements notiert werden:

```
<script>console.log("Hello World!");</script>
```

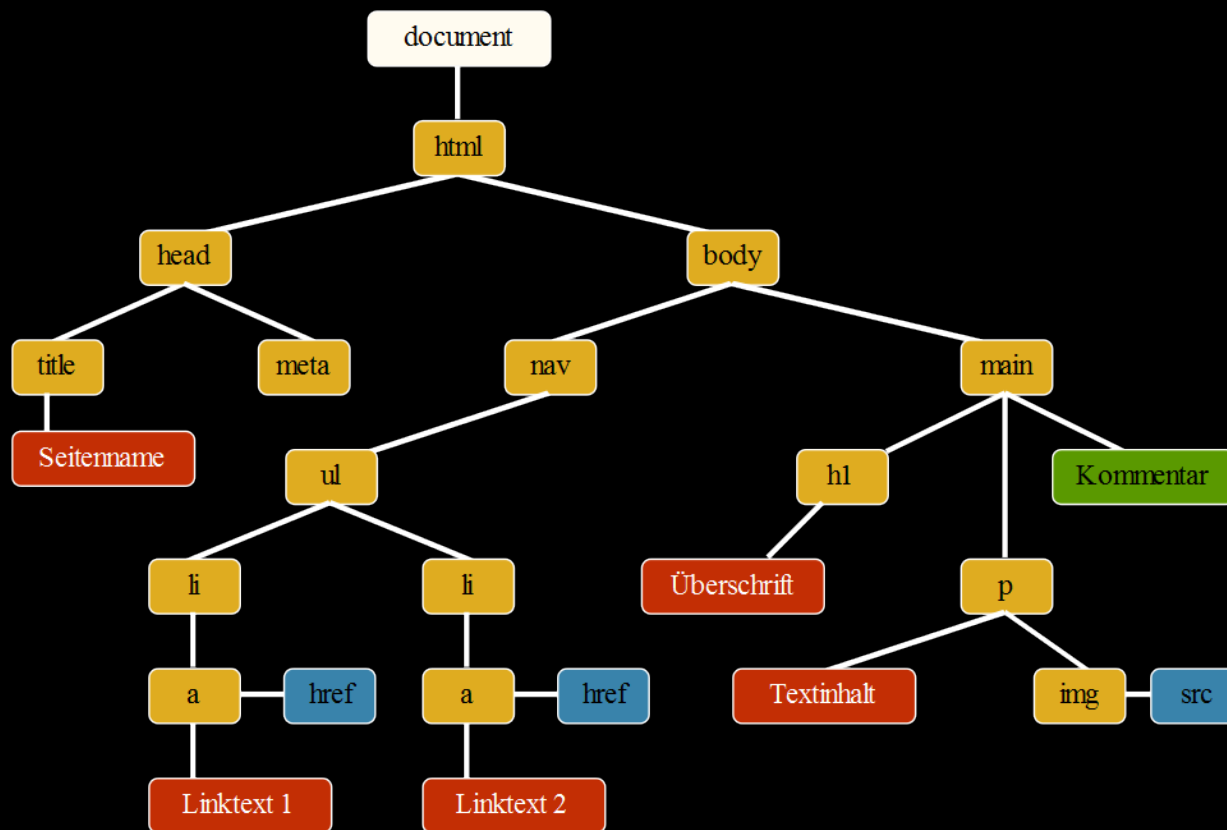
- JavaScript-Code kann in einer eigenen Datei liegen (Dateiendung `.js`) und im HTML (in `body` oder `head`) eingebunden werden:

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

- Vorteil – wie bei CSS; v.a. bessere Wartbarkeit

# Document Object Model (DOM)

Alle Elemente innerhalb des HTML-Dokuments werden durch das sogenannte DOM in JavaScript als ineinander verschachtelte Objekte verfügbar gemacht.



# DOM: Elemente und Knoten

Alle Elemente innerhalb des DOM kennen folgende Methoden (Funktionen), um schnell an darin enthaltene Knotenpunkte heranzukommen:

<code>getElementById("ID")</code>	-	Gibt den Knotenpunkt mit id ID zurück
<code>getElementsByName("name")</code>	-	node-list aller Knoten mit diesem name
<code>getElementsByTagName("p")</code>	-	node-list aller enthaltenen p-Elemente
<code>querySelector(selector)</code>	-	Erstes Element, das CSS-Selektor findet
<code>querySelectorAll(selector)</code>	-	node-list aller Elemente zum CSS-Selektor

Auch neue Knoten können erzeugt und eingefügt werden:

<code>document.createElement("tagName")</code>	-	Erzeugt einen neuen Knoten
<code>knoten.appendChild(kindKnoten)</code>	-	kindKnoten an Knoten anhängen
<code>knoten.insertBefore(kindKnoten, Referenz)</code>	-	so.o. mit: vor Referenz anhängen

# DOM: Elemente

Daneben haben alle DOM-Elemente die folgenden Methoden und Eigenschaften:

## Eigenschaften für Elementknoten

- `Element.accessKey`
- `Element.attributes`
- `Element.classList`
- `Element.className`
- `Element.clientHeight`
- `Element.clientWidth`
- `Element.dir`
- `Element.id`
- `Element.innerHTML`
- `Element.lang`
- `Element.namespaceURI`

- `Element.offsetHeight`
- `Element.offsetWidth`
- `Element.offsetLeft`
- `Element.offsetParent`
- `Element.offsetTop`
- `Element.scrollHeight`
- `Element.scrollLeft`
- `Element.scrollTop`
- `Element.scrollWidth`
- `Element.style`
- `Element.tabIndex`
- `Element.tagName`
- `Element.title`

## Methoden für Elementknoten

- `Element.animate()`
- `Element.closest()`
- `Element.getAttribute()`
- `Element.getAttributeNode()`
- `Element.getBoundingClientRect()`
- `Element.focus()`
- `Element.getElementsByTagName()`
- `Element.hasAttribute()`
- `Element.hasAttributes()`
- `Element.matches()`
- `Element.removeAttribute()`
- `Element.removeAttributeNode()`
- `Element.scrollIntoView()`
- `Element.setAttribute()`
- `Element.setAttributeNode()`

Mit der Eigenschaft `Element.innerHTML` kann HTML-Code eingefügt/verändert werden.



# DOM: Knoten

Daneben haben alle DOM-Knoten die folgenden Methoden und Eigenschaften:

## Eigenschaften für alle Knoten

- `Node.childNodes`
- `Node.firstChild`
- `innerText`
- `Node.lastChild`
- `childNodes.nextElementSibling`
- `Node.nextSibling`
- `Node.nodeName`
- `Node.nodeType`
- `Node.nodeValue`
- `Node.ownerDocument`
- `Node.parentElement`
- `Node.parentNode`
- `Node.previousSibling`
- `Node.previousElementSibling`
- `Node.textContent`

## Methoden für alle Knoten

- `Node.appendChild()` fügt ein neues Kindelement hinzu
- `Node.cloneNode()`
- `Node.compareDocumentPosition()`
- `Node.contains()`
- `Node.hasChildNodes()`
- `Node.insertBefore()`
- `Node.isDefaultNamespace()`
- `Node.isEqualNode()`
- `Node.normalize()`
- `Node.removeChild()`
- `Node.replaceChild()`

# Ist das jetzt schon interaktiv?



# DOM: Events

Interaktiv wird JavaScript erst, wenn wir auf Ereignisse reagieren, die bei der Bedienung der Webseite durch den Anwender passieren.

Dazu lauschen wir mittels EventListnern darauf, ob bestimmte Events an einem Knoten ausgelöst werden, und führen im Fall des Falles eine Funktion aus. Beispiel:

```
Node.addEventListener("load",function(event) {  
    console.log(event);  
    console.log("Das Element "+this+" wurde vollständig geladen!");  
});
```

Wichtige Events: (Vollständige Liste: <https://wiki.selfhtml.org/wiki/JavaScript/DOM/Event/%C3%9Cbersicht> )

- |           |   |   |
|-----------|---|---|
| change    | - | (Wert-)Änderung, vor allem bei Formularelementen            |
| click     | - | Beim Anklicken (alternativ: mouseup / mousedown)            |
| focus     | - | Beim Fokussieren (alternativ: focusin / focusout)           |
| load      | - | Element ist vollständig geladen und steht zur Verfügung     |
| mouseover | - | Beim überfahren mit der Maus (alternativ: mousein/mouseout) |
| scroll    | - | Beim Scrollen   |

# Tipps für JS-Manipulation der Seite

- Trotz Verfügbarkeit im DOM: Änderungen am Style-Objekt eines Nodes vermeiden. Meist besser: Klassen dynamisch hinzufügen und entfernen, Darstellung mit CSS regeln:

```
element.classList.add(className)
```

```
element.classList.remove(className)
```

```
element.classList.toggle(className)
```

- Skript erst nach `document.onload` ausführen; alle Anweisungen in einen `load`-Event-Handler schreiben
- Variablen nicht global definieren (`"var"`), sondern ein einzelnes globales Objekt definieren (z.B. `var myScript = {}`) und alle Funktionen, Variablen, ... dort ablegen: `myScript.variable = "x";`

# Noch Fragen?

JavaScript im SELFHTML-Wiki:

<https://wiki.selfhtml.org/wiki/JavaScript>

