

EINE EINFÜHRUNG IN PROGRAMMIERUNG VON  
WEBAPPLIKATIONEN MIT JAVASCRIPT.

JAVASCRIPT KOMPAKT

# DIE GESCHICHTE VON JAVASCRIPT

# BRENDAN EICH

- entwickelt 1995 die Sprache Mocha/JavaScript für den Netscape Navigator 2.0



# JAVASCRIPT

- Sun Microsystems und Netscape kooperieren - Ziel: Javaapplets werden mit LiveScript gesteuert.
- LiveScript wird in JavaScript 1.0 umbenannt, Netscape stellt LiveConnect zur Verfügung.



# ECMA - EUROPEAN COMPUTER MANUFACTURER ASSOCIATION

- Die ECMA übernimmt die Standardisierung des allgemeinen Teils von Javascript:
- Die einfachen Variabtentypen 'number', 'string' und 'boolean' werden genau beschrieben, deren Deklaration und Verhaltensweisen.
- Ebenso die komplexen Typen 'array', 'object' und 'function'.
- Dazu kommen einige Funktionsobjekte/-konstruktoren: Date, Math, RegExp, JSON
- Die Methode 'var' wird Pflicht, um Variablen zu deklarieren, Die 'use strict' Klausel trennt ECMA-konformes Scripting von 'freiem' Schreiben.

- Heute gelten die ECMA Standards 5.1 und 6.0+ (ES2015+).
- ES 6.0 ergänzt zum Beispiel 'class', 'private', 'public', 'protected' und weitere OO-Muster, aber auch Ausdrücke aus anderen Sprachen

# DIE MOZILLA FOUNDATION: DOM - JAVASCRIPT

- Die Mozilla Foundation als Nachfolger der ehemaligen Netscape Entwicklergruppe entwickelt den DOM-Part von Javascript, der für das Verhalten im Browser verantwortlich ist und implementiert dies in seinen Browser 'Firefox'
- Darin befinden sich alle DOM-Objekte und Methoden, wie 'window', 'document', 'navigator' etc. Diese werden von allen Browserherstellern weitestgehend übernommen.
- Mozilla entwickelt das NICHT-DOM-Objekt console zur Ausgabe von Werten und Typen in der Browserconsole. (Firebug war auch die erste Konsole für das Debugging im Browser).

- Microsoft entwickelte hier teilweise eigene Lösungen, die besser zu Architektur der Microsoft-Software passen: zum Beispiel das ActiveXObject, das bei Mozilla XMLHttpRequest heisst.
- Jscript, VBScript waren Microsofts frühe Implementationen der Javascript Idee. Heute unterstützt Microsoft Standard Javascript, entwickelt aber mit Typescript eine verbesserte Notationsweise.

# DIE JAVASCRIPT ENGINES IN DEN BROWSERN

- Jeder Browser verwendet eine eigene Engine zur Interpretation und Kompilierung von Javascript.
- Chrome, Safari und Opera verwenden eine Version der Opensource Javascript Engine V8. Microsoft hat Chakra entwickelt und verwendet diese ab der Versionen 10 des Internet Explorer.

- Die Browser implementieren teilweise zusätzliche Objekte und Methoden, so gibt es zum Beispiel in Googles Chrome ein 'chrome' Objekt, das die Grundlagen für Chrome-Apps liefert.

# BEKANNTEN ANWENDUNGEN VON ~~JAVASCRIPT~~ ECMASCRIPT AUSSERHALB DES BROWSERS

- Adobe Flash
- Adobe PDF
- Nodejs

# JAVASCRIPT IN HTML EINBINDEN

# WO WIRD JAVASCRIPT EINGEBUNDEN?

- Im Allgemeinen wird Javascript heute am Ende des Body Elementes der HTML Datei.
- Das stellt einen reibungslosen Ladevorgang des DOM und CSS sicher, bevor mit dem Parsen und Kompilieren von Javascript begonnen wird.
- Skripte, die nicht auf das DOM zugreifen und nicht sehr lang sind, werden im Body positioniert, (z.B. das Google Analytics Skript).

# JAVASCRIPT INCLUDES

Das Einbinden externer Javascript-Dateien hat einige Vorteile:  
Es muss nur eine (einige) Datei aktualisiert werden, alle  
Seiten nutzen automatisch das (geänderte) Skript.  
Die HTML Seiten sind übersichtlicher und bleiben sauber;

```
<html>
<head> ... </head>
<body>
  ...
  <script src="_scripts/javascript.js" ... ></script>
</body>
</html>
```

# ASYNC

```
<script async src="...">
```

Das Script wird asynchron zum Rest der Seite ausgeführt. Es startet, wenn es geladen ist.

# DEFER

```
<script defer src="...">>
```

Das Script wird in der Ladereihenfolge ausgeführt.  
Deferred Scripts laufen vor allen anderen geladenen Skripten.

# CHARSET

```
<script ... charset=„UTF-8“></script>
```

Das charset Attribut spezifiziert das Character Encoding der inkludierten Datei.

Es wird benötigt, wenn die inkludierte Datei einen anderen Zeichensatz verwendet, als die HTML Datei.

# DIE SPRACHELEMENTE VON JAVASCRIPT

# KOMMENTARZEICHEN

```
// slashslash für einzeiligen Kommentar
/*
  slashstar
  für einen Block
  Kommentar
*/
```

# VERWENDUNG VON JAVADOC-KOMPATIBLEN KOMMENTAREN IST GUT FÜRS TEAMWORK

```
/* vim: set expandtab tabstop=4 shiftwidth=4 softtabstop=4: */
<太后
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * written in Javascript 1.7, jQuery 1.7.2
 *
 * @package      myApplication
 * @author       Michael Reichart <reichart@michaelreichart.de>
 * @author       Another Name
 * @copyright   1999–2019 Michael Reichart
 * @license     http://www.michaelreichart.de/license/1.txt
 * @version     SVN: $Id$
 * @link        http://host.net/package/myApplication
 * @since      File available since Release 1.0.0
 * @deprecated File deprecated in Release 2.0.0
 */
```

# STANDARDKOMMENTARE FÜR FUNKTIONEN

```
/**  
 * Short description for Method  
 *  
 * Long description for method (if any)...  
 *  
 * @author      Michael Reichart  
 * @author      Your Name  
 * @version     1.0.0  
 * @since       Method available since Release 1.0.0  
 * @deprecated  File deprecated in Release 2.0.0  
 *  
 * @param type $varName  
 * @returns type  
 */
```

# VARIABLENNAMEN

```
var  
  a, _b, $c,  
  _, $,  
  a1, b_3, c$, myCamelCaseVariablesName,  
  x = null,  
  __ia__is_ie7_askjeu = false;
```

Konstruktorfunktionen beginnen mit einem Grossbuchstaben.

```
function Auto () { ... }  
var myAuto = new Auto();
```

# FUNKTIONEN UND FUNKTIONALE OBJEKTE

```
function log(msg) {  
  console.log('log: ' + msg);  
}
```

```
var log = function (msg) {  
  console.log('log: ' + msg);  
}
```

# NUMBERS UND NUMERISCHE LITERALE

```
var n1 = 9.81;    // dezimal  
var n2 = 0xa3;    // hexadezimal  
var n3 = 073;     // oktal
```

```
.01024e4  
1.024e+3  
10.24E2  
102.4E+1  
1024.e0  
1024.00  
1024  
10240e-1
```

# METHODEN FÜR NUMERISCHE WERTE

```
var a = 4  
  
a.toExponential()  
a.toFixed()  
a.toLocaleString()  
a.toPrecision()  
a.toString()  
a.valueOf()
```

# NAN - NOT A NUMBER

Ist das Ergebnis von nicht definierten oder fehlerbehafteten Rechenoperationen.

NaN ist giftig: Jede arithmetische Operation, die mit einem NaN rechnet, wird NaN als Ergebnis liefern.

NaN ist mit nichts anderem vergleichbar, auch mit sich selbst nicht.

```
NaN === NaN // false
```

```
NaN !== NaN // true
```

# STRINGS UND STRING.LENGTH

- Eine Folge von 0 oder mehr 16 bit Unicode Buchstaben (UCS-2)
- Die "length" Eigenschaft gibt die Anzahl der 16-bit Zeichen in einem String zurück.
- Erweiterte Zeichen werden als 2 Zeichen gezählt.

# + VERBINDEN VON STRINGS

- + kann Strings verbinden oder addieren.

```
'$' + '1' + '2' === '$12';  
'$'.concat('1').concat('2');
```

```
'$' + 1 + 2 = ,$12';  
1 + 2 + ,"$" = ,3$"'
```

# ZAHLEN IN EINEN STRING KONVERTIEREN

```
str = num.toString();
```

```
str = String(num);
```

# SPRINGS IN EINE ZAHL KONVERTIEREN

Mit der Number Funktion:

```
num = Number(str);
```

Mit dem + Präfixoperator:

```
num = +str;
```

Mit der parseInt/parseFloat Funktion:

```
num = parseFloat(str);
```

```
num = parseInt(str);
```

# DIE PARSEINT FUNKTION

Die Konvertierung mit prassend endet beim ersten Nicht-Ziffern-Zeichen.

```
parseInt(str, 10);  
parseInt("12em", 10) === 12;
```

Das Anhängsel (10) stellt das Dezimalsystem ein und sollte immer verwendet werden.

```
parseInt("08")      === 0 (ES3)  
parseInt("08", 10) === 8
```

# STRING METHODEN

charAt()  
charCodeAt()  
compareLocale()  
concat()  
indexOf()  
lastIndexOf()  
localeCompare()  
match()  
replace()  
search()

slice()  
split()  
substring()  
toLocaleLowerCase()  
toLocaleUpperCase()  
toLowerCase()  
toString()  
toUpperCase()  
trim()  
valueOf()

# BOOLEAN UND FALSEY VALUES

```
false === false;  
  
    0 === false;  
    '' === false;  
    null === false;  
undefined === false;  
    Nan === false;  
  
var variable;  
    variable == undefined == false;  
  
Alle anderen Werte sind truthy!
```

# BOOLEAN UND FALSEY VALUES

```
1 === '1'  
1 !== '1'  
1 === 1
```

```
0 == false;  
'' == false;  
null == false;  
undefined == false;  
Nan == false;
```

```
var variable;  
variable == undefined == false;
```

Alle anderen Werte sind truthy!

# JAVASCRIPT OBJECT NOTATION - JSON

Ein Objekt ist eine dynamische Sammlung von Eigenschaften. Jede Eigenschaft hat eine String als Namen. Der String ist unique!  
Die Attribute sind public !

```
var tier = {  
    art      : "Hund",  
    beine    : 2,  
    fluegel  : 0,  
    bellen   : function(){  
        log('wau!');  
    }  
};  
  
tier.beine = 4;  
tier.schwanz = 1 ;  
delete tier.fluegel;
```

# ARRAYS

```
var array = [true, 'Zwei', 3];  
array[0]; array[1];  
array.push('new value')  
for (i = 0; i < a.length; i++) {  
    var val = a[i];  
}  
array[array.length] = 'new value';
```

# ARRAY METHODS

concat  
every  
filter  
forEach  
indexOf  
join  
lastIndexOf  
map  
pop  
push

reduce  
reduceRight  
reverse  
shift  
slice  
some  
splice  
toString  
unshift

# SORT

```
var n = [4, 8, 15, 16, 23, 42];
n.sort();
// n is [15, 16, 23, 4, 42, 8]
```

# ELEMENTE EINES ARRAY LÖSCHEN

```
myArray = ['a', 'b', 'c', 'd'];

delete myArray[1];
// ['a', undefined, 'c', 'd']

myArray.splice(1, 1);
// ['a', 'c', 'd']
```

# NULL UND UNDEFINED

- **null** ist der Standard-**leer**-wert für Variablen und Parameter.
- Es ist der Wert für fehlende Member in Objekten.
- **undefined** wird zurückgegeben, wenn ein Objekt nicht existiert.
- null und undefined sind falsy values.

# CALL BY REFERENCES OR BY VALUE?

# CALL BY REFERENCE

- Objekte können als Argumente in Funktionen übergeben werden. Und können von Funktionen als Returnwert zurückgegeben werden.
- Objekte werden dabei als Referenz behandelt.

# CALL BY VALUE

- Die einfachen Variablentypen `string`, `number`, und `boolean` werden als Wert übergeben.

# WERTE ODER WERTE/TYPEN-VERGLEICH?

Der `==` Operator vergleicht Objektreferenzen, und nicht deren Werte. Nur wenn beide Operanden ein und dasselbe Objekt sind.

```
1 === true -> false  
1 == true -> true
```

# OPERATOREN

# OPERATOREN

- Arithmetisch + - \* / %
- Vergleichend == != < > <= >= === !==
- Logisch && || !
- Bitweise & | ^ >> >>> <<
- Ternary ?:

# DEFAULT OPERATOR

```
function (arg) {  
  var value = arg || default;  
}
```

# BITWEISE

& | ^ >> >>> <<

Die bitweisen Operatoren konvertieren die Operanden zu einer 32-bit vorzeichenfähigen Integerzahl und liefern das Ergebnis wieder als 64-bit Fließkommazahl ab.

# STATEMENTS

# STATEMENTS

if  
switch  
while  
do  
for

break  
continue  
return

try ... catch/throw

# FOR STATEMENT

Iteriert durch alle Elemente eines Arrays:

```
for (var i = 0; i < array.length; i++) {  
    // within the loop,  
    // i is the index of the current member  
    // array[i] is the current element  
}
```

# FOR IN STATEMENT

Iteriert durch alle Elemente eines Objektes:

```
for (name in object) {  
    if (object.hasOwnProperty(name)) {  
        // within the loop,  
        // name is the key of current member  
        // object[name] is the current value  
    }  
}
```

# SWITCH STATEMENT

- Mehrwegverzweigung
- Der Switch-Wert muss keine Zahl, sondern kann auch ein String sein.
- Die einzelnen Cases können Ausdrücke und Programmzeilen beinhalten.
- Gefährlich: Cases fallen in den nächsten Case durch, solange der Case nicht durch ein break unterbrochen und beendet wird.

# SWITCH STATEMENT

```
switch (expression) {  
    case ';':  
    case ',':  
    case '.':  
        punctuation();  
        break;  
    default:  
        noneOfTheAbove();  
}
```

# WHILE () {}

```
while (condition) {  
    doSomethingAwesome();  
}  
  
var a = 0;  
while(a<4) {  
    doSomethingAwesome();  
    a++;  
}
```

# DO {} WHILE ()

```
do {  
    doSomethingAwesome();  
} while (condition)  
  
var a = 0;  
do {  
    doSomethingAwesome();  
    a++;  
} while(a<4)
```

# TRY STATEMENT

The **try** statement lets you test a block of code for errors.

The **catch** statement lets you handle the error.

The **throw** statement lets you create custom errors.

The **finally** statement lets you execute code,  
after try and catch, regardless of the result.

# THROW STATEMENT

```
throw new Error(reason);  
  
throw {  
  name: exceptionName,  
  message: reason  
};
```

# TRY STATEMENT

```
try {  
    addAlert("Welcome guest!");  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.message;  
}
```

# TRY AND THROW

```
try {  
    throw "myException"; // Fehler wird ausgelöst  
}  
catch (e) {  
    // Anweisungen für jeden Fehler  
    logMyErrors(e); // Fehler-Objekt an die Error-Funktion  
    geben  
}
```

# TRY STATEMENT

```
function myFunction() {  
    var message, x;  
    message = document.getElementById("p01");  
    message.innerHTML = "";  
    x = document.getElementById("demo").value;  
  
    try {  
        if(x == "") throw "empty";  
        if(isNaN(x)) throw "not a number";  
        x = Number(x);  
        if(x < 5) throw "too low";  
        if(x > 10) throw "too high";  
    }  
    catch(err) {  
        message.innerHTML = "Input is " + err;  
    }  
}
```

# TRY STATEMENT

```
try {  
    try_Anweisungen  
}  
[catch (Fehler_var_2) {  
    catch_Anweisungen_2  
}]  
[finally {  
    finally_Anweisungen  
}]
```

# TRY STATEMENT

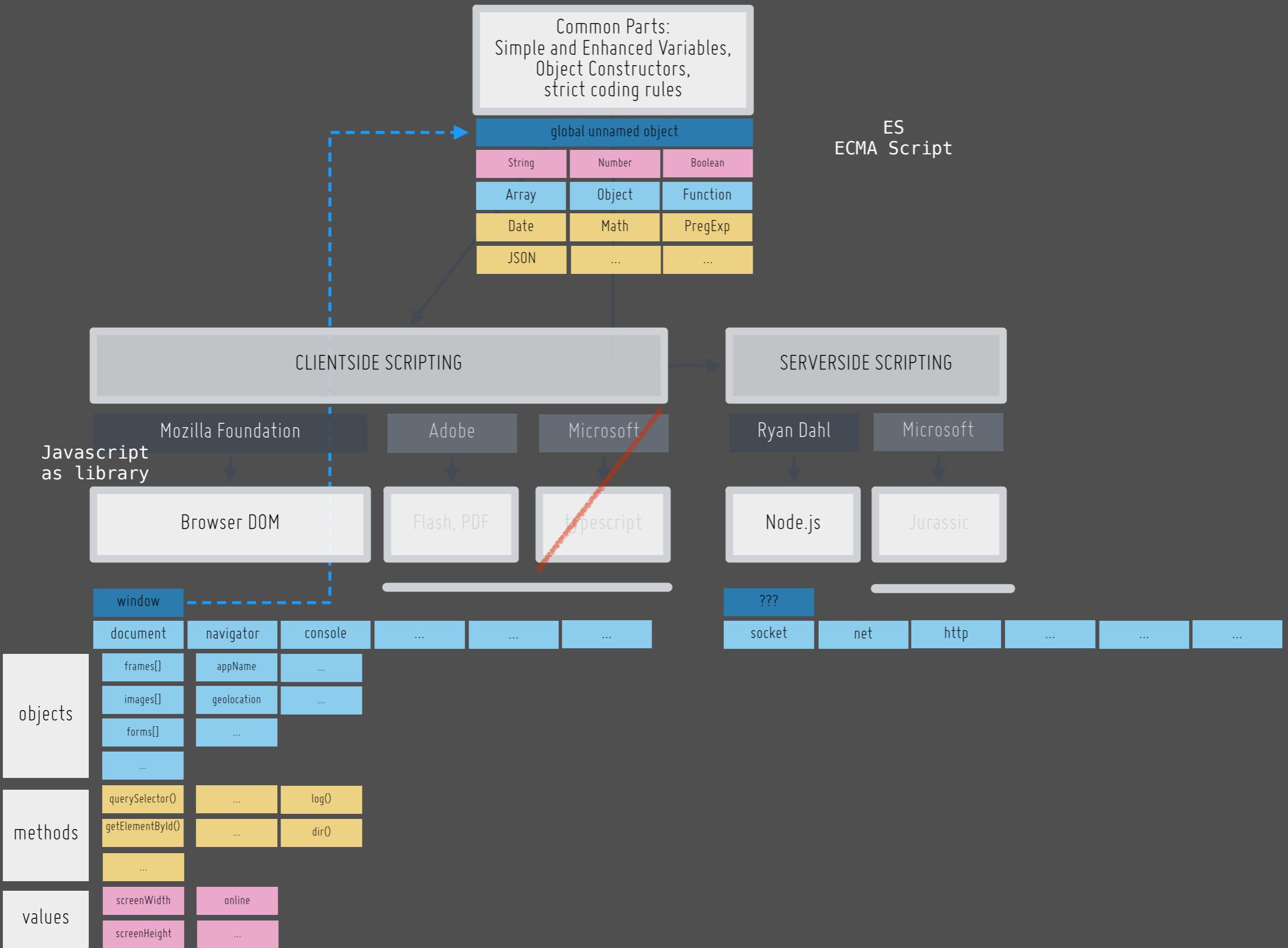
Die JavaScript Implementierung kann folgende Exception names ausgeben:

- 'Error'
- 'EvalError'
- 'RangeError'
- 'SyntaxError'
- 'TypeError'
- 'URIError'

# DIE KOMPONENTEN VON JAVASCRIPT

# JAVASCRIPT

- Javascript ist im Grunde keine Programmiersprache (das ist ECMA Script), sondern eine Objektbibliothek..
- -> Objektbasiert, nicht klassenbasiert.
- Objekt bedeutet hier entweder ein Objekt aus dem Document Object Model, sprich ein HTML-Knoten (Node), oder ein selbst erzeugtes.



# ZWEI BESONDERHEITEN VON JAVASCRIPT

- „var“ erzeugt Funktionsscopes.
- „let“ (ES 6) erzeugt Controlscopes
- Objekte sind public.
- Werte haben Typen, es gibt aber keine implizite Typenüberprüfung.
- Typen können sich ändern.

# ECMA OBJEKT-PROTOTYPEN

- Das **namenlose globale Objekt**, das alle Variablen und Objekte enthält. (**function () { ... }()**)
- **Object** als allgemeiner Prototyp, von dem alle Objekte abgeleitet sind
- **Number** als Prototyp für Zahlen (64-Bit-Gleitkommazahlen gemäß IEEE 754)
- **String** als Prototyp für Zeichenketten (16-bit UCS-2 Zeichenketten)
- **Boolean** als Prototyp für boolesche Werte (true, false)
- **Function** als Prototyp für Funktionen
- **Array** als Prototyp für Arrays (numerisch indiziertes Objekt)

# TYPEN IN JAVASCRIPT

number	a = 1;	(kein int, float, double)
string	a = 'Hallo'; a="H";	(no char);
boolean	a = 3>4; a=false;	
object	a = {key:"value"};	
array	a = [true, 2, "drei"];	
function	a = function () {};	

# UNIFIED NUMBER TYPE

```
a = 0.1; b = 0.2; c = 0.3;  
(a + b) + c === a + (b + c) // false
```

Die Teilung von zwei Integerzahlen kann einen Nicht-Integerwert hervorbringen.

$10 / 3 = 3.333333333333335$

Unäre Operatoren können Strings in Zahlen konvertieren.

`+ "42" = 42`

# + ODER CONCAT

Addition und Stringverkettung?

$$3 + 4 = 7$$

$$'$' + 3 + 4 = '$34'$$

$$3 + '4' = 34$$

$$3 + 4 + '5' = 75$$

# JSON - JAVASCRIPT OBJECT NOTATION

```
var obj = {  
    key_1 : 'Value 1',  
    key_2 : 42,  
    key_3 : false,  
    key_4 : [true, 2, 'drei'],  
    key_5 : { ... },  
    key_6 : function () { ... }  
}
```

# JSON NACH ALLGEMEINEN REGELN

```
{  
    "key_1" : "Value 1",  
    "key_2" : 42,  
    "key_3" : false,  
    "key_4" : [true, 2, "drei"],  
    "key_5" : { ... },  
}  
  
[  
    ["Michael", "Cologne", "michael@zenbox.de"],  
    ["Paula", "Stuttgart", "paula@zenbox.de"]  
]
```

# EINGEBAUTE OBJEKTE, DIE VON ECMASCRIPT DEFINIERT WERDEN.

- **Math** stellt Konstanten und Methoden für mathematische Operationen bereit. Math kann nicht als Konstruktor dienen.
- **Date** für Operationen mit Daten bzw. Zeitpunkten und Datumsformaten
- **RegExp** für reguläre Ausdrücke
- **JSON** zur Verarbeitung von Objekten

# DIE OBJEKT-BIBLIOTHEK DES BROWSERS

window, document, navigator, screen, history, localStorage,  
console, XMLHttpRequest, ...

Das window-Objekt selbst ist dabei de facto das globale Objekt, indem einer Variablen window das globale Objekt zugewiesen wurde.

```
(function () {  
    var window = this;  
}());
```

# SCOPES - GÜLTIGKEITSBEREICHE VON VARIABLEN

# FUNKTIONEN BILDEN SCOPES

```
fn = function (arg) {  
    var member = arg || null;  
}
```

# ERZEUGEN GLOBALE VARIABLEN

```
<script>
  a = 2;                      // global: window.a
  var c = 3;                    // global: window.c
  function f () {
    b = 3;                      // auch global!! window.b
    var d = 4;
  }
</script>
```

# LOKALE VARIABLEN - "VAR" INNERHALB VON FUNKTIONEN

```
function f () {  
    "use strict";  
    var a;      // Lokale Variable mit var  
    ...;  
}
```

# LET ERZEUGT SEIT ES6+ KONTROLSCOPE

```
for (let i=0; i<10; i++) {  
    . . .  
}  
  
console.log(i); // -> i = undefined
```

# USE STRICT

# GENERAL RESTRICTIONS IN STRICT MODE

- 1. No Undeclared Variables**
2. No deleting of a variable or a function
3. No duplicating of a parameter name
4. The eval Method cannot alter scope
5. The eval and arguments cannot be used as identifiers
6. The with statement is not allowed
7. Writing to a read-only property is not allowed
8. Adding a new property to a non-extensible object is not allowed
9. No octal numbers
10. Keywords reserved for future are not allowed

<https://www.tutorialrepublic.com/javascript-tutorial/javascript-strict-mode.php>

FUNKTIONEN, DIE SICH SELBST AUSFÜHREN.  
SIE WERDEN VOR ALLEN ANDEREN FUNKTIONEN INITIALISIERT UND KÖNNEN NEBENBEI  
DAZU VERWENDET WERDEN, PRIVATE KEYS UND METHODEN ZU VERWENDEN.

# IMMEDIATE FUNCTIONS

# IIFE - DIE IMMEDIATE INVOKED FUNCTION EXPRESSION

```
!(function () {}()); oder !(function () {})();
```

Das erste Klammerpaar ist ein Rangfolge Operator. Er zieht die Berechnung des Ausdrucks in der Klammer "nach vorne". Der Compiler berechnet den Ausdruck unmittelbar nach dem Kompilieren.

Das zweite Klammerpaar ist der Aufruf des ersten.

# AUFBAU EINER IMMEDIATE FUNCTION

```
var extArgs;  
  
( function (intArgs) { ... } (extArgs));  
  
/*  
innerhalb der Immediate Function wird in der Regel eine  
anonyme Funktion platziert, die durch die Immediate Function  
ausgeführt wird. Ihr werden die Argumente übergeben.  
*/
```

# IMMEDIATE FUNCTION MIT RÜCKGABEWERT

```
var module;

module = (function () {
    var _obj = {};
    ...
    return _obj;
}());

/*
Da die Funktion sofort ausgeführt ist, gibt sie anstelle eines
Funktionszeigers ein Ergebnis zurück.
*/
```

# EIN OBJEKT MIT PUBLIC UND PRIVATE KEYS.

```
var animal = (function (type, legs, wings, sound) {  
  
    var _type  = type  || 'Hund',  
        _legs   = legs  || '4',  
        _wings  = wings || '0',  
        _sound   = sound || 'wau';  
  
    function _getType () { return _type; }  
  
    function _setType (value) { _type = value; }  
    return ({  
        getType : _getType,  
        setType : _setType,  
        sound   : _sound  
    });  
}());  
  
// Getter!  
console.log( animal.getType() );
```

# JAVASCRIPT ALS TEIL VON HTML 5

# THE CONCEPT OF HTML5 AS LANGUAGE PACKAGE FOR USER INTERFACES

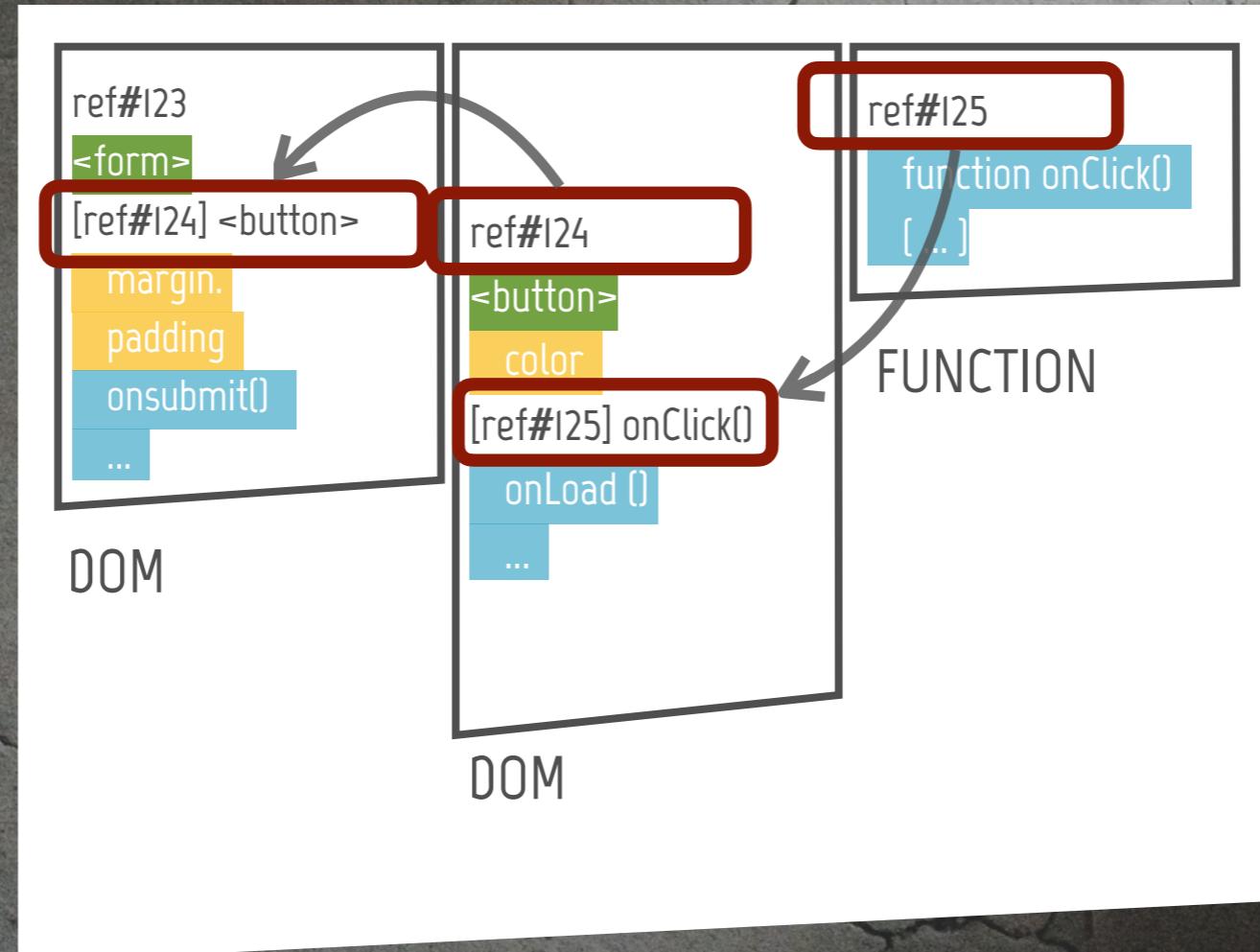
structure  
objects with  
HTML

set properties  
with  
STYLESHEETS

build  
functionality  
with  
JAVASCRIPT

# Pointer Call by reference

MEMORY

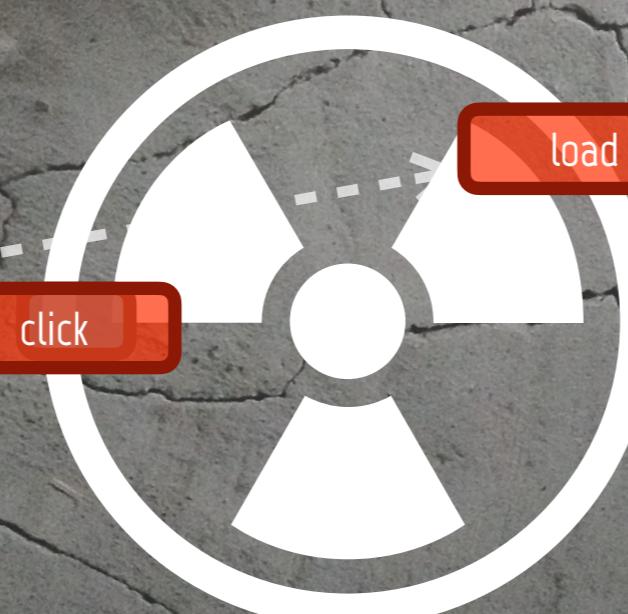


# ASYNCHRONIZITÄT. WARUM IST JAVASCRIPT ANDERS?

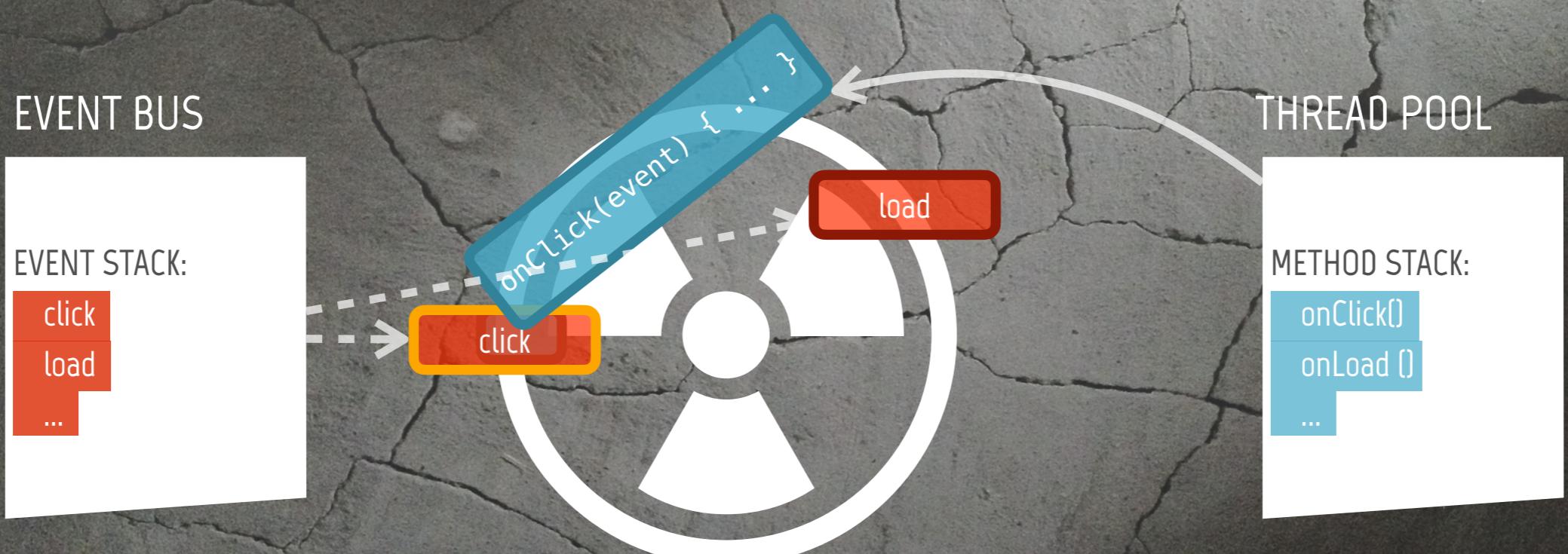
EVENT BUS



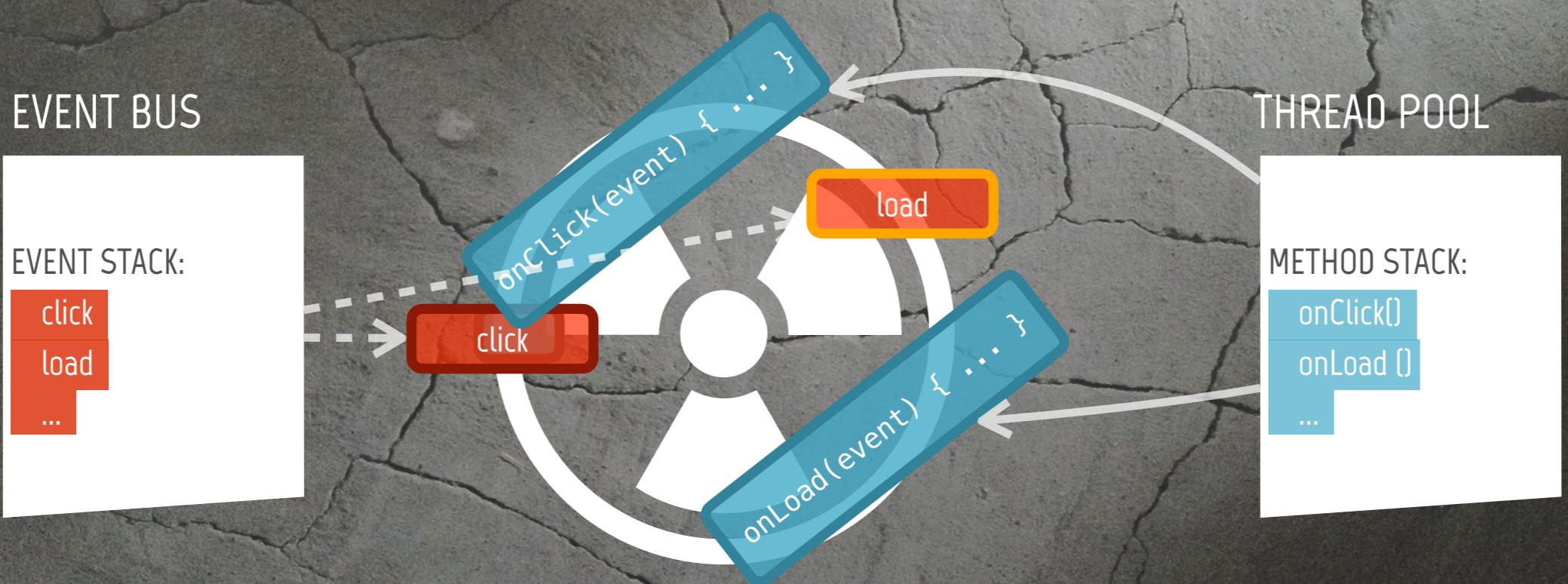
THREAD POOL



Single Thread  
Async Function Processing  
Non-Blocking I/O



Single Thread  
Async Function Processing  
Non-Blocking I/O

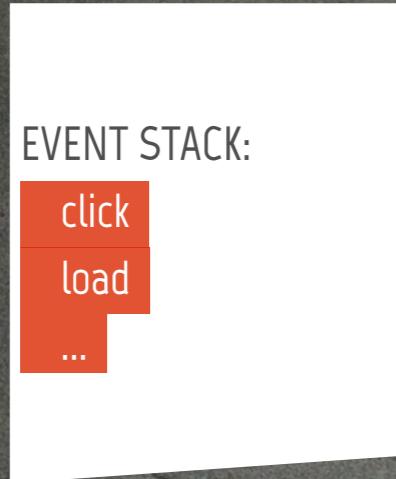


Single Thread  
Async Function Processing  
Non-Blocking I/O

onClick(event) { ... }

onLoad(

EVENT BUS



THREAD POOL

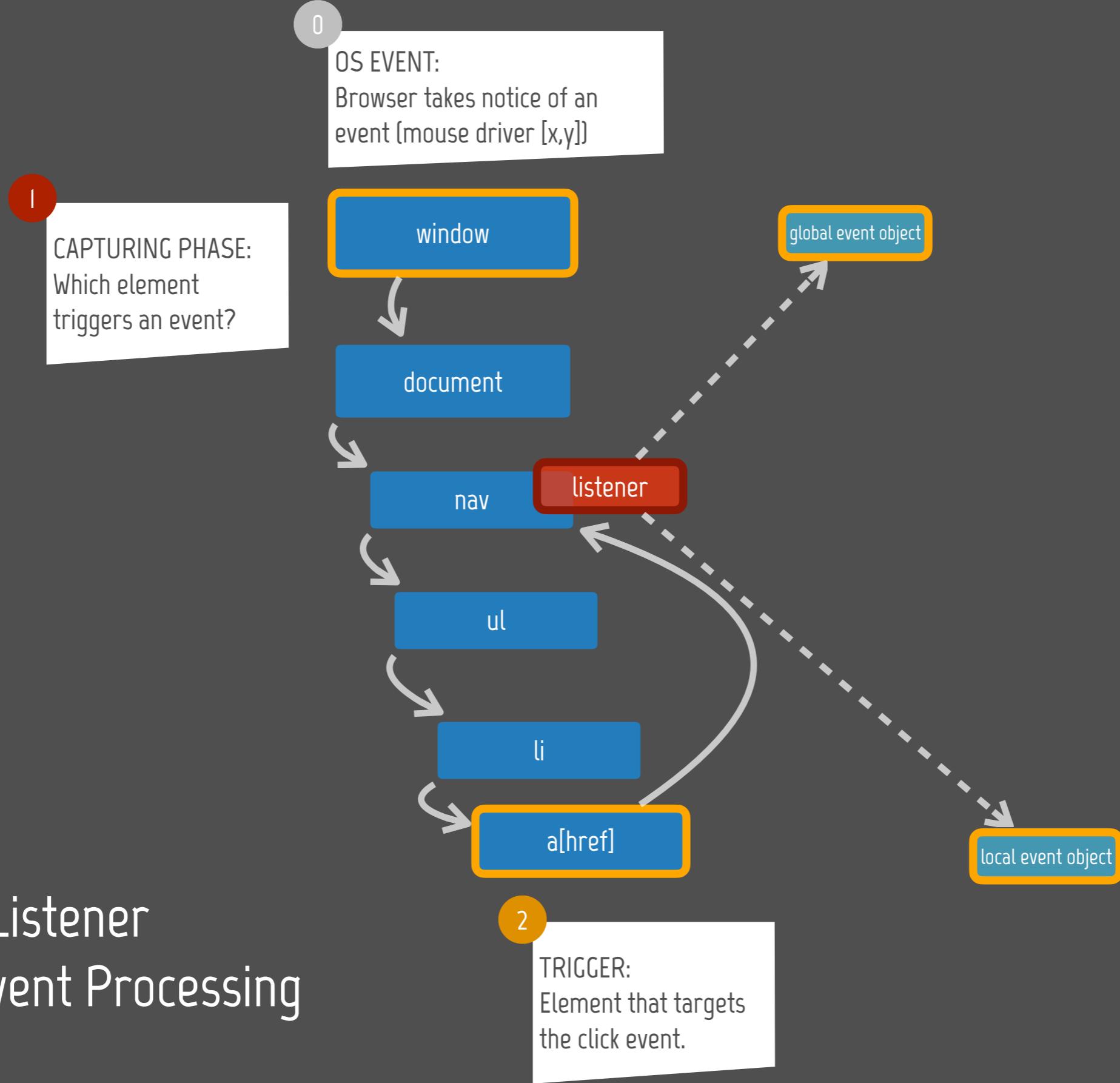


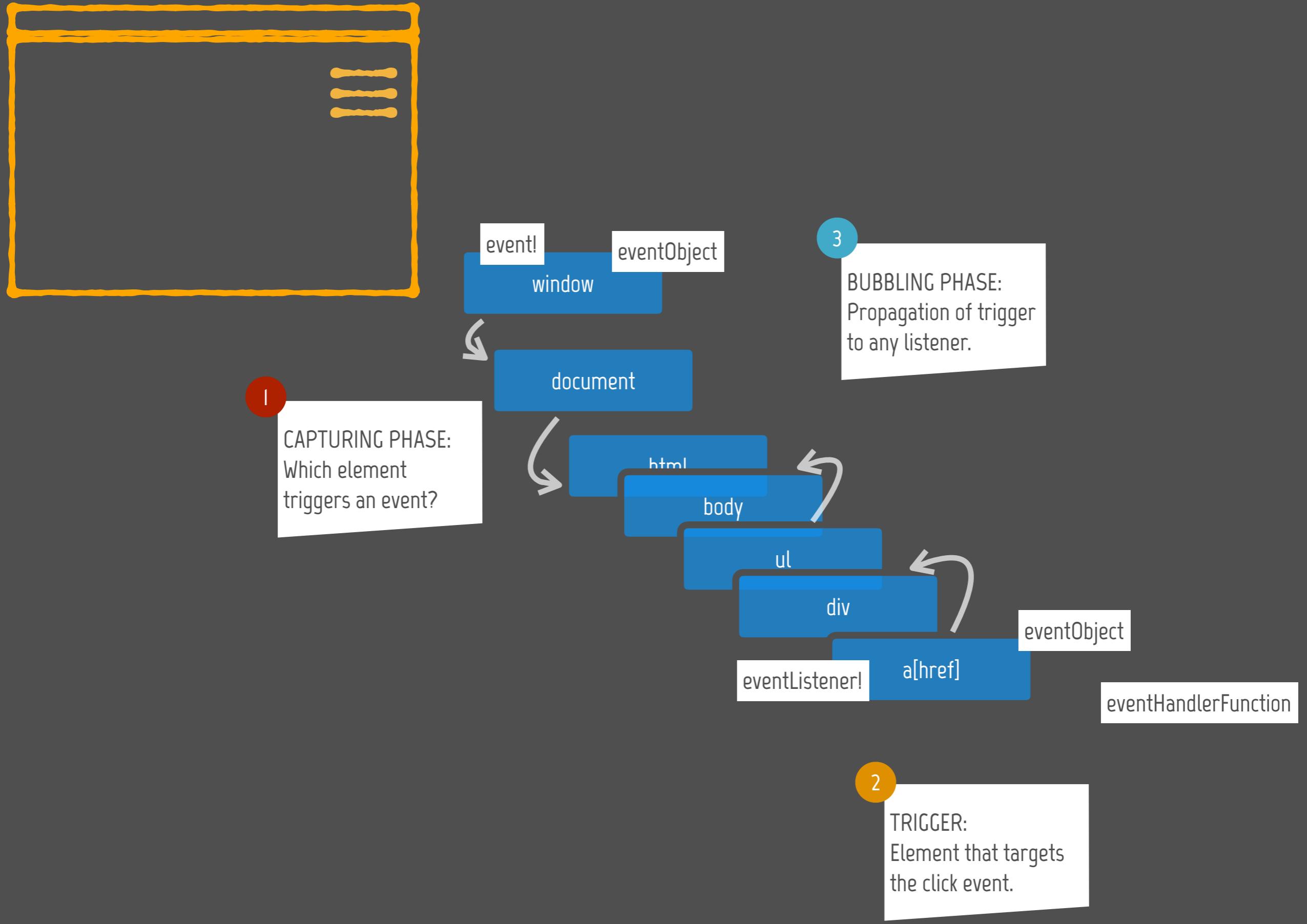
Single Thread  
Async Function Processing  
Non-Blocking I/O

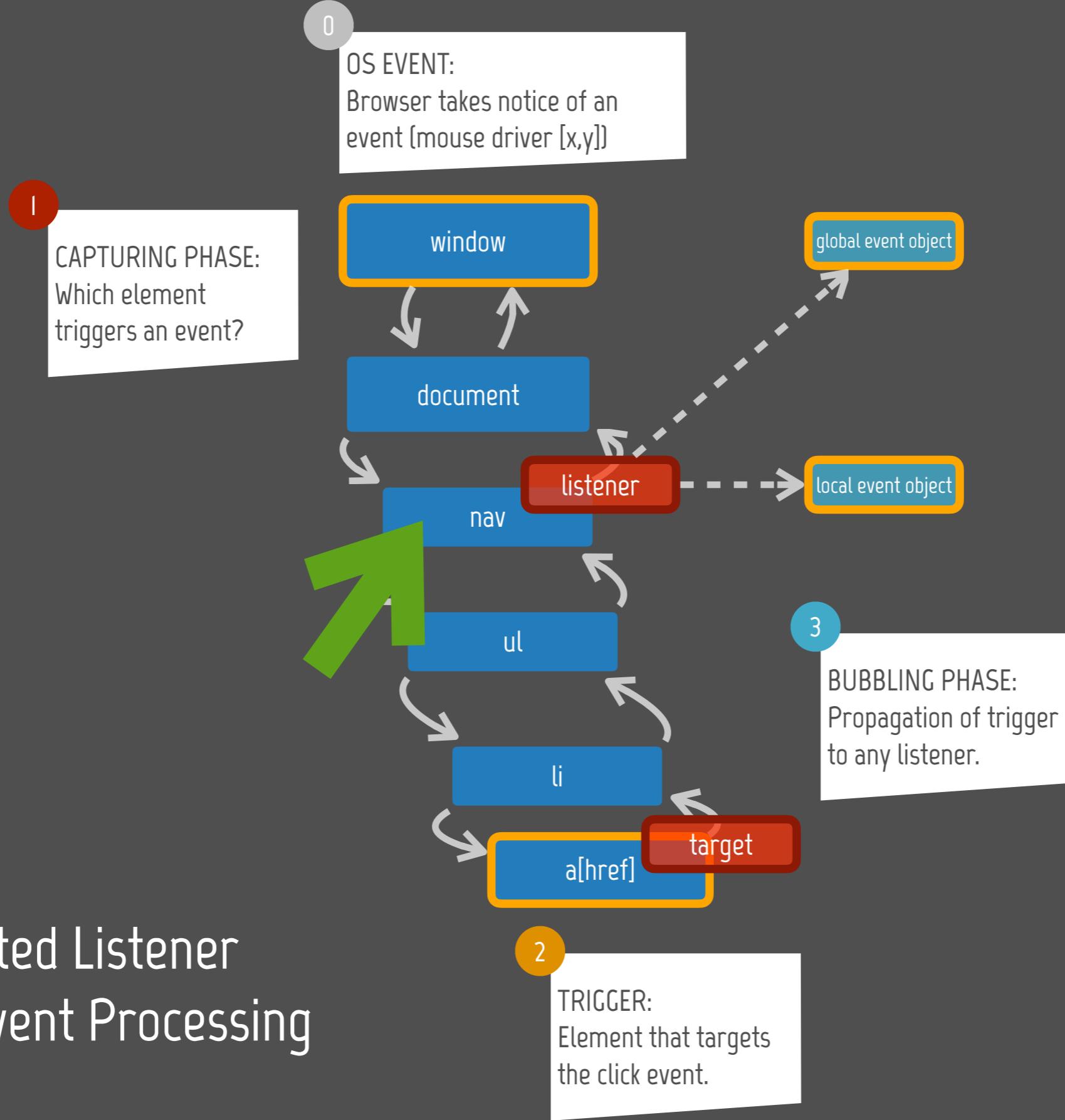
onLoad(event) { ... }



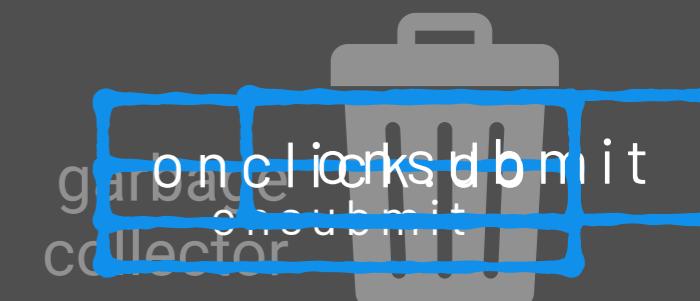
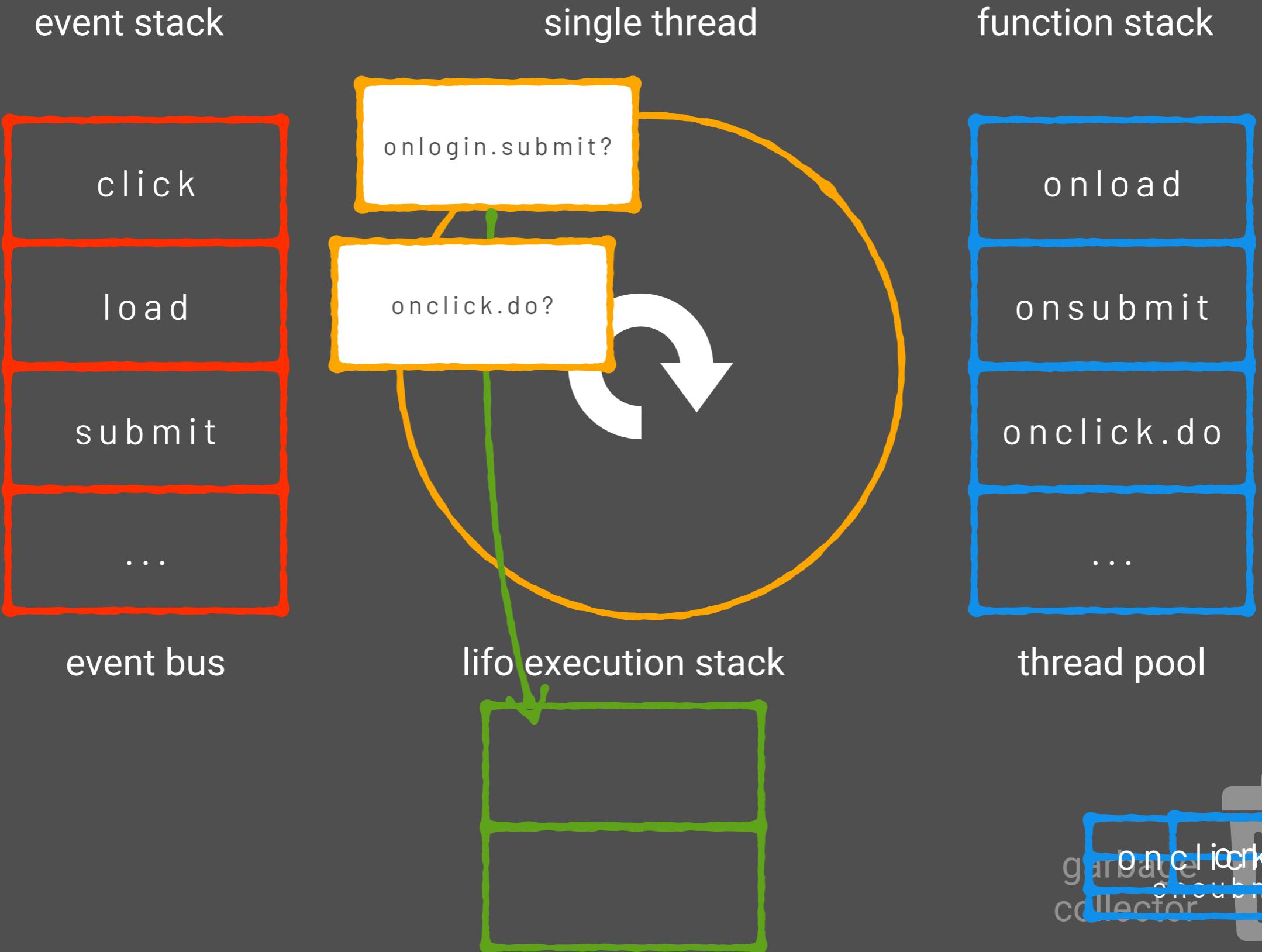
# JAVASCRIPT EVENTS







# EVENT PROCESSING



# EVENTS

- Alle Elementobjekte und weitere Objekte besitzen die Methode addEventListener.
- Ein Eventlistener enthält einen Ereignistyp, ein Elementobjekt und eine Handler-Funktion.

```
element.addEventListener( "event",
                        handlerfunktion,
                        capturing);
```

"event" ist ein String und enthält den Ereignistyp:  
"click", "mouseover", "load", "submit" ...

Das Handler–Funktionsobjekt, das ausgeführt werden soll.

Der dritte Parameter bestimmt, für welche Event–Phase der Handler registriert werden soll. Die Werte sind true oder false.

false steht für die Bubbling–Phase  
(sollte als Standard verwendet werden).  
true für die Capturing–Phase.

# EIN BEISPIEL

```
window.addEventListener("load", start, false);

function start () {
    var pElement = document.getElementById("interaktiv");
    pElement.addEventListener("click", klickverarbeitung,
false);
}
function klickverarbeitung () {
    document.getElementById("interaktiv").innerHTML +=
    " Das ist dynamisch generierter Text.";
}
```

# EVENT-HANDLER ENTFERNEN: REMOVEEVENTLISTENER

Um die mit addEventListener registrierten Handler wieder zu entfernen, gibt es die Methode removeEventListener. Die Methode erwartet dieselben Parameter, die addEventListener beim Registrieren bekommen hat.

```
function setzen () {  
    pElement.addEventListener("click", onClick);  
    pElement.addEventListener("click", onClick2);  
}  
  
function beenden () {  
    pElement.removeEventListener("click", onClick);  
    pElement.removeEventListener("click");  
}
```

# EIGENE EVENTS

```
// Einen neuen Event Typ anmelden
var e = new Event('send');

// Einen neuen CustomEvent Typ anmelden
var myEventType = new CustomEvent('mySend',
    {detail:'some data'});

// Den Event abschicken
anyObject.dispatchEvent(myEventType);

// Der Eventlistener, wie gewohnt
domObj.addEventListener('mySend', function (event) { ... });
```

# EVENTLISTENER IN IE < 9

```
window.attachEvent("onload", start);

function start () {
    var pElement = document.getElementById("interaktiv");
    pElement.attachEvent("onclick", klickverarbeitung);
}

function klickverarbeitung () {
    document.getElementById("interaktiv").innerHTML +=
        " Das ist dynamisch generierter Text.";
}
```

# EVENTS IN IE < 9 ENTFERNEN

```
function beenden () {  
    pElement.detachEvent("onclick", klickverarbeitung);  
}
```

# EINE BROWSERÜBERGREIFENDE EVENTFUNKTION

```
function addEvent (obj, type, fn) {  
    if (obj.addEventListener) {  
        obj.addEventListener(type, fn, false);  
    } else if (obj.attachEvent) {  
        obj.attachEvent('on' + type, function () {  
            return fn.call(obj, window.event);  
        });  
    }  
}
```

Eine bessere unter:

[http://therealcrisp.xs4all.nl/upload/addEvent\\_dean.html](http://therealcrisp.xs4all.nl/upload/addEvent_dean.html)

# DAS EVENTOBJEKT MIT PREVENTDEFAULT

```
function zeigeVollbild (eventObjekt) {  
    // Browserübergreifender Zugriff auf das Event-Objekt  
    if (!eventObjekt) eventObjekt = window.event;  
  
    // Existiert die Methode preventDefault? Dann rufe sie auf.  
    if (eventObjekt.preventDefault) {  
        // W3C-DOM-Standard  
        eventObjekt.preventDefault();  
    } else {  
        // Andernfalls setze returnValue  
        // Microsoft-Alternative für Internet Explorer < 9  
        eventObjekt.returnValue = false;  
    }  
};
```

## THE EVENT OBJECT

```
event  
target  
delegateTarget  
  
which  
type  
meta-/shift-/alt-/ctrlKey  
  
pageX  
pageY  
  
preventDefault()  
stopPropagation()  
  
...
```

## AN ONCLICK EVENT HANDLER

```
fn = {  
  onclick : function (event) {  
    event.preventDefault();  
    event.stopPropagation();  
    url = $(event.target).attr('href');  
    fn.loadContentOf(url)  
  },  
  loadContentOf : function (url) {...}  
}
```

USE `event.target`, NOT `this`!

# ECMA OBJEKTE

# MATH

# DIE METHODEN DES MATH OBJECTS

```
var a = Math.abs(3.14152);  
var u = 4 * Math.PI * r;
```

abs()  
acos()  
asin()  
atan()  
atan2()  
ceil()  
cos()  
exp()  
floor()

log()  
max()  
min()  
pow()  
random()  
round()  
sin()  
sqrt()  
tan()  
PI

# DATE

# DER DATE KONSTRUKTOR

```
const heute = new Date(); // aktuelles Datum und aktuelle Zeit  
- 20.11.2019, 07:47:06
```

```
now.getDate(); // Tage des Monats (1–31)
```

Erzeugt eine Date-Instanz, die einen Zeitmoment enthält.

Datumsobjekte basieren auf einem Wert in Millisekunden seit dem 1. Januar 1970, UTC.

# DER DATE KONSTRUKTOR

```
var today = new Date();
var birthday = new Date("December 17, 1995 03:24:00");
var birthday = new Date("1995-12-17T03:24:00");
var birthday = new Date(1995,11,17);
var birthday = new Date(1995,11,17,3,24,0);
```

# MIT DATUM RECHNEN

```
let date1 = new Date (2000,0,1);
let date2 = new Date (2000,0,1);

if (date1 === date2 ) { ... immer false!!!}

if (date1.getTime() === date2.getTime()) {
  1333609301900 === 1333609301900 // true !!!
}
```

# METHODEN DES DATE OBJEKTES

Date.now()	Returns the numeric value corresponding to the current time - the number of milliseconds elapsed since 1 January 1970 00:00:00 UTC.
Date.parse()	Parses a string representation of a date and returns the number of milliseconds since 1 January, 1970, 00:00:00, local time.
Date.UTC()	Accepts the same parameters as the longest form of the constructor (i.e. 2 to 7) and returns the number of milliseconds since 1 January, 1970, 00:00:00 UTC.

# GETTER DER DATE INSTANZEN

.getDate()	Returns the day of the month (1-31) for the specified date according to local time.
.getDay()	Returns the day of the week (0-6) for the specified date according to local time.
.getFullYear()	Returns the year (4 digits for 4-digit years) of the specified date according to local time.
.getHours()	Returns the hour (0-23) in the specified date according to local time.
.getMilliseconds()	Returns the milliseconds (0-999) in the specified date according to local time.
.getMinutes()	Returns the minutes (0-59) in the specified date according to local time.
.getMonth()	Returns the month (0-11) in the specified date according to local time.
.getSeconds()	Returns the seconds (0-59) in the specified date according to local time.

# GETTER DER DATE INSTANZEN

.getTime()	Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC (negative for prior times).
.getTimezoneOffset()	Returns the time-zone offset in minutes for the current locale.
.getUTCDate()	Returns the day (date) of the month (1-31) in the specified date according to universal time.
.getUTCDay()	Returns the day of the week (0-6) in the specified date according to universal time.
.getUTCFullYear()	Returns the year (4 digits for 4-digit years) in the specified date according to universal time.
.getUTCHours()	Returns the hours (0-23) in the specified date according to universal time.
.getUTCMilliseconds()	Returns the milliseconds (0-999) in the specified date according to universal time.
.getUTCMinutes()	Returns the minutes (0-59) in the specified date according to universal time.

# GETTER DER DATE INSTANZEN

.getUTCMonth()	Returns the month (0-11) in the specified date according to UTC.
.getUTCSeconds()	Returns the seconds (0-59) in the specified date according to UTC.
.getYear()	Returns the year (usually 2-3 digits) in the specified date according to UTC. Use getFullYear() instead.
.getHours()	Returns the hours (0-23) in the specified date according to UTC.
.getMinutes()	Returns the minutes (0-59) in the specified date according to UTC.
.getSeconds()	Returns the seconds (0-59) in the specified date according to UTC.
.getMilliseconds()	Returns the milliseconds (0-999) in the specified date according to UTC.

# SETTER

- `setDate()`
- `setFullYear()`
- `setHours()`
- `setMilliseconds()`
- `setMinutes()`
- `setMonth()`
- `setSeconds()`
- `setTime()`
  
- `setUTCDate()`
- `setUTCFullYear()`
- `setUTCMonth()`
- `setUTCSeconds()`
- `setUTCHours()`
- `setUTCMilliseconds()`
- `setUTCMinutes()`
- `setYear()`

# CONVERSION GETTER

- `toDateString()`
- `toISOString()`
- `toJSON()`
- `toGMTString()`
- `toLocaleDateString()`
- `toLocaleFormat()`
- `toLocaleString()`
- `toLocaleTimeString()`
- `toSource()`
- `toString()`
- `toTimeString()`
- `toUTCString()`
- `valueOf()`

# REGEXP

# REGEXP

- Ein regulärer Ausdruck spezifiziert eine Syntax eines Suchausdrucks für Texte.
- Reguläre Ausdrücke folgen einer Syntax.
- RegExp erlaubt keine Leerzeichen oder Kommentare  
→ Sie sind schwer zu lesen.

# BEISPIELE FÜR REGULÄRE AUSDRÜCKE

Ein einfacher regulärer Ausdruck in Javascript:

```
var reg = /ab+c/;  
var reg = new RegExp("ab+c");
```

```
var myArray = reg.exec("cdbbdbbz");
```

Ein komplexer Ausdruck für Email-Adressen:

```
var reg = /^[a-zA-Z0-9][\w\.-]*@(?:[a-zA-Z0-9][a-zA-Z0-9_-]+\.)+[A-Z,a-z]{2,5}$/;
```

```
var reg = /(\w+)\s(\w+)/;  
var str = "Jonas Schmidt";  
var newstr = str.replace(reg, "$2, $1");
```

# BEISPIELE FÜR REGULÄRE AUSDRÜCKE

```
var reg = /\(?(\d{3})\)?([-\/\.\.])\d{3}\1\d{4}/;

function testInfo(phoneInput){

    var OK = reg.exec(phoneInput.value);

    if (!OK)
        window.alert(RegExp.input + " ist keine Telefonnummer mit Vorwahl!");
    else
        window.alert("Danke! Ihre Telefonnummer ist " + OK[0]);
}
```

# METHODEN DIE REGULÄRE AUSDRÜCKE VERWENDEN

- exec Eine Methode von RegExp, die Suche nach einer Übereinstimmung in einer Zeichenkette durchführt. Sie gibt ein Array mit den Übereinstimmungen zurück.
- 
- test Eine Methode von RegExp, die eine Zeichenkette auf eine Übereinstimmung überprüft. Sie gibt true oder false zurück.
- 
- match Eine Methode von String, die eine Suche nach einer Übereinstimmung in einer Zeichenkette durchführt. Sie gibt ein Array zurück oder null bei keinem Treffer.
- 
- search Eine Methode von String, die eine Zeichenkette auf eine Übereinstimmung überprüft. Sie gibt den Index der Übereinstimmung zurück oder -1 bei keinem Treffer.
- 
- replace Eine Methode von String, die eine Suche nach einer Übereinstimmung in einer Zeichenkette durchführt und die Übereinstimmungen durch eine andere Zeichenkette ersetzt.
- 
- split Eine Methode von String, die anhand eines regulären Ausdrucks oder einer festen Zeichkette eine Zeichenkette trennt und die Teile als Array zurückgibt.

# BESONDERE ZEICHEN UND AUSDRÜCKE

//	Start – Ende eines RegExp
\	escaped Slash (\*, ...)
^	zu Beginn
\$	am Ende
.	Wildcard, beliebiges Zeichen
	Alternative

# ZEICHENBEREICHE

- [] Set/Bereich von Zeichen [0-9], [A-Za-z0-9]  
findet jedes dieser Zeichen
- [xyz] Zeichenbereich, der alle notierten Zeichen erfasst
- [^xyz] Erfasst keines der notierten Zeichen

# BESONDERE ZEICHEN UND AUSDRÜCKE

\b	Wortgrenze (Anfang/Ende)
\s	Jedes Leerzeichen, Tabulator oder Umbruch
\d	Jede Ziffer
\w	Jedes Vorschubzeichen
\n	Zeilenvorschub
\w	Jeder Buchstabe, inkl. _

# MENGENANGABEN

+	einen oder mehrere der vorangehenden Zeichengruppe
?	Keinen oder einen ...
*	keinen oder mehr ...
{x}	x mal ...
{x,}	x mal oder mehr ...
{x,y}	x bis y mal ...

# GRUPPIERUNGEN

(...) Gruppe; höchstens 9 Gruppen

?:(...) Ausschlußgruppe

Parameter, auch Kombinationen sind möglich

/.../g globale Suche

/.../i unabhängig von Groß- und Kleinschreibung

/.../m Multiline

# JSON

# DAS JSON - OBJEKT

```
var obj = JSON.parse(text [, reviver])
```

Schreibt ein als JSON-String gegebenes Objekt in ein Objekt in den Speicher. So kann ein per Ajax gelieferter JSON-String weiterverarbeitet werden.

```
var string = JSON.stringify(obj [, replacer] [, space])
```

Schreibt ein im Speicher liegendes Objekt in einen String um. Der String kann nun zum Server geschickt oder gespeichert werden.

# DIE BROWSEROBJEKTE IN JAVASCRIPT

# BROWSER OBJEKTE UND UNTEROBJEKTE

## Window

document

anchors

DOM

forms

elements

options

images

links

history

location

Seite

- Referenz auf das globale anonyme Objekt
- enthält das DOM (Document Object Modell)
- Ein Array mit allen anchor-Elementen aus dem DOM
- Ein Array mit allen form-Elementen aus dem DOM
- Ein Array mit allen Elementen einer Form
- 
- Ein Array mit allen img-Elementen aus dem DOM
- Ein Array mit allen Links aus dem DOM
- Ein Array mit allen besuchten URIs
- Ein Objekt mit Informationen zur aktuellen Seite

## Navigator

mimeType

plugins

- Ein Objekt mit Browserinformationen
  - Ein Array mit allen mimeType
  - Ein Array mit allen Plugin Informationen

## Screen

- Ein Objekt mit Eigenschaften zum Bildschirm

# DAS WINDOW OBJEKT

Das windows Objekt liefert Informationen über das aktuell angesprochene Fenster (das das Dokument enthält).  
Es kennt u.a. folgende Eigenschaften:

`window.frames`  
`window.location`  
`window.name`  
`window.opener`  
`window.parent`  
`window.status`

und einige mehr.

# DAS WINDOW OBJEKT HAT METHODEN

Methoden sind Fähigkeiten (was kann ...) eines Objektes.

```
windows.alert()  
windows.blur()  
windows.clearTimeout()  
windows.close()  
windows.confirm()  
windows.focus()  
windows.prompt()  
windows.scroll()  
windows.setTimeout()
```

# AUßerdem reagiert es auf Events

Das Window Objekt kennt u.a. folgende Eventhandler:  
onBlur = ...                            onFocus = ...  
onLoad = ...                            onUnload = ...

Zum Beispiel stellt der folgende Event sicher, dass das, was innerhalb der anonymen Funktion steht, erst ausgeführt wird, wenn das Dokument vollständig geladen ist.  
`window.onload = function () {`

`...`

`}`

# DAS NAVIGATOR OBJEKT

Das Navigator Objekt liefert Informationen über den verwendeten Browser.

`navigator.appName`  
`navigator.appVersion`  
`navigator.appCodeName`  
`navigator.mimeTypes`  
`navigator.plugins`  
`navigator.userAgent`  
`navigator.plugins`

<http://de.selfhtml.org/javascript/objekte/index.htm>

– JAVASCRIPT OBJEKTREFERENZ

# DOM MANIPULATION

# DOKUMENTENKNOTEN (DOM NODES) SIND OBJEKTE

```
// Document Object Model! – DOM

<html>
  <head>...</head>
  <body>
    <div id="box"> ...</div>
  </body>
</html>

→ document.getElementById('box');
```

# IDENTIFIZIEREN VON KNOTEN

```
var node = document.getElementById('content');

.getElementsByClassName("green")
.getElementsByTagName("p")

.querySelectorAll("p.green")
.querySelector()
```

# MANCHE ELEMENTE ERLAUBEN DEN DIREKTEN ZUGRIFF

```
// access for <html>
document.documentElement.getAttribute("lang"));

// access for <body>
document.body.style.background = "yellow";

// access for <head>
alert(document.head.firstChild.nodeName)
```

# ERST ERZEUGEN, DANN EINFÜGEN ...

- AUCH TEXTKNOTEN SIND KNOTEN

```
var head      = document.createElement('h1');
var headText = document.createTextNode('Lorem ipsum dolor.');

head.appendChild(headText);

document.body.appendChild(head);
```

# NEUE ELEMENTE ERZEUGEN BESTEHENDE AUSTAUSCHEN

```
var parentElem = document.getElementById("main");
var oldPara = document.getElementById("hint");

// Creating new element
var newPara = document.createElement("p");
var newContent = document.createTextNode("This is new");
newPara.appendChild(newContent);

// Replacing old paragraph with newly created paragraph
parentElem.replaceChild(newPara, oldPara);
```

# APPENDING SIBLINGS BEFORE

```
let currentDiv = document.getElementById("main"),
    newDiv = document.createElement("div"),
    newContent = document.createTextNode("Lorem ipsum dolor.");

newDiv.appendChild(newContent);

document.body.insertBefore(newDiv, currentDiv);
```

# APPENDING SIBLINGS OR CHILDREN

```
let main = document.getElementById("main"),
    paragraph = createElement('p'),
    lorem = createTextNode('Lorem ipsum dolor.');

paragraph.appendChild(lorem);

// Inserting HTML just before the element itself,
// as a previous sibling
main.insertAdjacentHTML('beforebegin', paragraph);

// Inserting HTML just inside the element,
// before its first child
main.insertAdjacentHTML('afterbegin', paragraph);

// Inserting HTML just inside the element,
// after its last child
main.insertAdjacentHTML('beforeend', paragraph);

// Inserting HTML just after the element itself,
// as a next sibling
main.insertAdjacentHTML('afterend', paragraph);
```

# ELEMENTE LÖSCHEN

```
var childElem = document.getElementById("hint");  
childElem.parentNode.removeChild(childElem);
```



# SETZEN VON ATTRIBUTEN

```
var head      = document.createElement('h1');
var headText = document.createTextNode('Lorem ipsum dolor.');

head.appendChild(headText);

head.setAttribute('id', 'myId')
head.classList.add('myClass')

document.body.appendChild(head);
```

# STYLES VERÄNDERN

# SETZEN UND LESEN VON STYLE PROPERTIES

```
var elem = document.getElementById("intro");

// Applying styles on element
elem.style.color      = "blue";
elem.style.fontSize   = "18px";
elem.style.fontWeight = "bold";

console.log( elem.style.fontWeight ); // "bold"

'border-left-style' -> 'borderLeftStyle'
```

# WINDOW.GETCOMPUTEDSTYLE()

```
<!DOCTYPE html>
<html>
<head>
<style type="text/css">
  #intro {
    font-weight: bold;
    font-style: italic;
  }
</style>
</head>
<body>
  <p id="intro" style="color:red; font-size:20px;">This is a paragraph.</p>
  <p>This is another paragraph.</p>

  <script>
var elem = document.getElementById("intro");

var styles = window.getComputedStyle(elem);

console.log(styles.getPropertyValue("color"));          // Outputs: rgb(255, 0, 0)
console.log(styles.getPropertyValue("font-size"));       // Outputs: 20px
console.log(styles.getPropertyValue("font-weight"));     // Outputs: 700
console.log(styles.getPropertyValue("font-style"));      // Outputs: italic
</script>
</body>
</html>
```

# CLASSES

```
var elem = document.getElementById("info");

elem.classList.add("hide"); // Add a new class
elem.classList.add("note", "highlight"); // Add multiple classes
elem.classList.remove("hide"); // Remove a class
elem.classList.remove("disabled", "note"); // Remove multiple classes

// If class exists remove it, if not add it
elem.classList.toggle("visible");

// Determine if class exist
if(elem.classList.contains("highlight")) {
    console.log("The specified class exists on the element.");
}
```

# DATENAUSTAUSCH UND KOMMUNIKATION ZWISCHEN SERVER UND CLIENT

# AJAX - ASYNCHRONOUS JAVASCRIPT AND XML

- AJAX ist eine standardisierte Methode, um per Javascript Daten vom Server anzufordern.
- AJAX basiert auf dem **XMLHttpRequest** Object.

```
let request = new XMLHttpRequest();

request.open(filename, 'get');
request.send();
-----
request.addEventListener('readystatechange', function()
{ ... });
```

# AJAX - ASYNCRONOUS JAVASCRIPT AND XML

```
xhr = new XMLHttpRequest();
xhr.addEventListener('readystatechange', onReadyStateChange);
xhr.open("GET", _file);
xhr.send();

function onReadyStateChange() {
    switch (xhr.readyState) {
        case 0:
            console.log('there is no request');
            break;
        case 1:
            console.log('request opened');
            break;
        case 2:
            console.log('request sent');
            break;
        case 3:
            console.log('response first part ...');
            break;
        case 4:
            console.log('response more parts and finished!');
            if (xhr.status === 200 || xhr.status === 304) {
                _data = JSON.parse(xhr.response);
            }
            break;
        default:
            console.log('something strange happened!');
            break;
    }
}
```

# FETCH() (ES 6+)

```
fetch('data.json')
  .then(function (response) {
    if (response.ok)
      return response.json();
    else
      throw new Error('Kurse konnten nicht geladen werden');
  })
  .then(function (json) {
    // Hier Code zum einarbeiten der Kurse in die Anzeige
  })
  .catch(function (err) {
    // Hier Fehlerbehandlung
  });

// fetch basiert auf Promises,
// die ebenfalls in ES 6+ eingeführt wurden
```

# FETCH() EINSTELLEN

```
fetch( url, {  
    method: 'POST', // *GET, POST, PUT, DELETE, etc.  
    mode: 'cors', // no-cors, *cors, same-origin  
    cache: 'no-cache', // *default, no-cache, reload, force-cache,  
    credentials: 'same-origin', // only-if-cac  
    headers: { // include, *same-origin, omit  
        'Content-Type': 'application/json' // 'application/x-www-form-urlencoded',  
    },  
  
    redirect: 'follow', // manual, *follow, error  
    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade,  
    // origin, origin-when-cross-origin,  
    // same-origin, strict-origin,  
    // strict-origin-when-cross-origin,  
    // unsafe-url  
  
    body: JSON.stringify(data) // body data type must match "Content-Type"  
    header  
})
```

[https://developer.mozilla.org/en-US/docs/Web/API/  
Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)

[https://dmitripavlutin.com/javascript-fetch-async-  
await/](https://dmitripavlutin.com/javascript-fetch-async-await/)

MEHR ÜBER FETCH()

# FORM DATA

# FORMDATA EINSAMMELN (ES6+)

```
let data = new FormData();

data.append("email" , document.querySelector("#login-
email").value);
data.append("password" , document.querySelector("#login-
password").value);

data.append("data" , "Wird noch mit dazugepackt");
```

# FORMDATA EINSAMMELN (ES6+)

```
// FormData.values () gibt einen Iterator zurück,  
// der über alle Werte des FormData-Objekts läuft.  
  
for (let value of formData.values()) {  
    console.log(value);  
}  
  
// FormData.entries () gibt Name und Wert zurück.  
  
for (let value of formData.entries()) {  
    console.log ("entries " + value[0] + " mit " + value[1]);  
}
```

# FORMDATA WEITERGEBEN (ES6+)

```
let
  form = document.getElementById('login'),
  formData = new FormData(form),
  request = new XMLHttpRequest();

request.addEventListener('readystatechange', function () {
  if (request.readyState !== 4) return false;
  console.log(request.responseText);
})

request.open('POST', 'login.php', true);
request.send(formData);
```

<https://thoughtbot.com/blog/ridiculously-simple-ajax-uploads-with-formdata>

# COOKIES

# WHAT IS A COOKIE?

- Cookies are data, stored in small text files, on your computer.
- When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.
- Cookies were invented to solve the problem "how to remember information about the user":
- When a user visits a web page, his name can be stored in a cookie.
- Next time the user visits the page, the cookie "remembers" his name.

- Cookies are saved in name-value pairs like:
- username=John Doe
- When a browser request a web page from a server, cookies belonging to the page is added to the request. This way the server gets the necessary data to "remember" information about users.

# CREATE A COOKIE WITH JAVASCRIPT

With JavaScript, a cookie can be created like this:

```
document.cookie="username=John Doe";
```

You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

```
document.cookie="username=John Doe; expires=Thu, 18 Dec 2013  
12:00:00 UTC";
```

With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.

```
document.cookie="username=John Doe; expires=Thu, 18 Dec 2013  
12:00:00 UTC; path=/";
```

# READ A COOKIE WITH JAVASCRIPT

With JavaScript, cookies can be read like this:

```
var x = document.cookie;
```

Note `document.cookie` will return all cookies in one string  
much like:

```
cookie1=value; cookie2=value; cookie3=value;
```

# CHANGE A COOKIE WITH JAVASCRIPT

With JavaScript, you can change a cookie the same way as you create it:

```
document.cookie="username=John Smith; expires=Thu, 18 Dec 2013  
12:00:00 UTC; path=/";
```

The old cookie is overwritten.

# DELETE A COOKIE WITH JAVASCRIPT

Deleting a cookie is very simple. Just set the expires parameter to a passed date:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970  
00:00:00 UTC";
```

Note that you don't have to specify a cookie value when you delete a cookie.

# A FUNCTION TO SET A COOKIE

First, we create a function that stores the name of the visitor in a cookie variable:

```
function setCookie(cname, cvalue, exdays) {  
    var d = new Date();  
    d.setTime(d.getTime()  
        + (exdays*24*60*60*1000));  
    var expires =  
        "expires="+d.toUTCString();  
    document.cookie = cname + "="  
        + cvalue + "; " + expires;  
}
```

# EXAMPLE EXPLAINED:

- The parameters of the function above are the name of the cookie (cname), the value of the cookie (cvalue), and the number of days until the cookie should expire (exdays).
- The function sets a cookie by adding together the cookiename, the cookie value, and the expires string.

# A FUNCTION TO GET A COOKIE

Then, we create a function that returns the value of a specified cookie:

```
function getCookie cname {  
    var name = cname + "=";  
    var ca = document.cookie.split(';' );  
    for(var i=0; i<ca.length; i++) {  
        var c = ca[i];  
        while (c.charAt(0)==' ') c = c.substring(1);  
        if (c.indexOf(name) != -1)  
            return c.substring(name.length,c.length);  
    }  
    return "";  
}
```

# FUNCTION EXPLAINED:

- Take the cookiename as parameter (cname).
- Create a variable (name) with the text to search for (cname + "=").
- Split document.cookie on semicolons into an array called ca (ca = document.cookie.split(';')).
- Loop through the ca array ( $i=0; i < ca.length; i++$ ), and read out each value c=ca[i]).
- If the cookie is found (c.indexOf(name) == 0), return the value of the cookie (c.substring(name.length,c.length)).
- If the cookie is not found, return "".

# A FUNCTION TO CHECK A COOKIE

Last, we create the function that checks if a cookie is set. If the cookie is set it will display a greeting. If the cookie is not set, it will display a prompt box, asking for the name of the user, and stores the username cookie for 365 days, by calling the setCookie function:

```
function checkCookie() {  
    var username=getCookie("username");  
    if (username!="") {  
        alert("Welcome again " + username);  
    }else{  
        username = prompt("Please enter your name:", "");  
        if (username != "" && username != null) {  
            setCookie("username", username, 365);  
        }  
    }  
}
```

# THINKING ON MODULES AND LIBRARIES

ECMA SCRIPT 5

# THE MODULE BLOCK PATTERN

# THE MODULE BLOCK PATTERN

```
(function () {
    'use strict';
    // -----
    var fn = {},
        a = 42;

    fn.log = function (message) {
        console.log(message);
    };

    fn.alert = function () {};

    window.myNamespace = window.myNamespace || {};
    window.myNamespace.myModuleName = fn;
    // -----
})();
```

# COOPERATION (DEPENDENCY INJECTION)

```
(function () {  
    // - - - - -  
    var fn = {};  
  
    fn.log = function (message) {  
        console.log(message);  
    };  
  
    console.log('Tut was in Module 2!');  
  
    window.app = window.app || {}; // Defaultoperator  
    window.app.myModule2 = fn;  
  
    window.app.myModule1.log('Aufruf an 1 aus 2');  
  
    // - - - - -  
}());
```

ECMA SCRIPT 5

# CONSTRUCTOR PATTERN

# OBJEKT AS CONSTRUCTOR

```
function Person(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
  
    this.fullName = function() {  
        return this.firstName + ' ' + this.lastName;  
    };  
}  
  
// Use it like this:  
var john = new Person('John', 'Doe');  
john.firstName;      // "John"  
john.fullName();    // "John Doe"  
  
john.firstName = 'John';  
john.fullName();    // John Doe
```

# KONSTRUKTOR OHNE THIS

```
function Person(firstName, lastName) {  
    var _firstName = firstName,  
        _lastName = lastName;  
  
    var my = {  
        firstName: _firstName,  
        lastName: _lastName  
    };  
  
    my.fullName = function() {  
        return _firstName + ' ' + _lastName;  
    };
```

# KONSTRUKTOR OHNE THIS

```
// Getter/setters

my.firstName = function(value) {
  if (!arguments.length) return _firstName;
  _firstName = value;

  return my;
};

my.lastName = function(value) {
  if (!arguments.length) return _lastName;
  _lastName = value;

  return my;
};

return my;
}
```

# KONSTRUKTOR OHNE THIS

```
// Use it like this:  
  
var chuck = Person('Chuck', 'Norris');  
  
chuck.firstName('Jackie');  
chuck.lastName('Chan');  
  
chuck.fullName(); // Jackie Chan
```

„<http://www.samselikoff.com/blog/some-Javascript-constructor-patterns/>“

# PROTOTYPES IN KONSTRUKTOREN

```
var Shipment = function (state, type) {  
    this.state = state || undefined;  
    this.type  = type  || undefined;  
  
    this.setState = function () {};  
};
```

# PROTOTYPES IN KONSTRUKTOREN

```
// Nachträglich hinzugefügte Methoden
Shipment.prototype.setState = function (value) {
    this.state = value;
};

Shipment.prototype.setType  = function (value) {
    this.type  = value;
};

Shipment.prototype.save      =
    function () {
        console.log( 'saving shipment '
                    + this.state + ', '
                    + this.type + '.');
    };
}
```

```
var shipment = [];  
  
shipment[0] = new Shipment(3, 1);  
shipment[1] = new Shipment();  
  
shipment[0].save();  
  
shipment[1].setState(4);  
shipment[1].setType(2);  
  
shipment[1].save();
```

ECMA SCRIPT 5

# INHERITANCE PATTERN

# INHERITANCE

```
var ShipmentRainbow =  
function (state, type, typeOfDoc, reference){  
    this.state      = state || undefined;  
    this.type       = type  || undefined;  
    this.typeOfDoc = typeOfDoc || 'HTML';  
    this.reference = reference || 'Hallo';  
};  
  
ShipmentRainbow.prototype = new Shipment();  
ShipmentRainbow.constructor = ShipmentRainbow;
```

# INHERITANCE

```
var shipmentRainbow = new  
ShipmentRainbow(3,1,'typeOfDocText','referenceText');  
  
shipmentRainbow.save();  
  
console.dir(shipmentRainbow);  
  
console.log( shipmentRainbow instanceof Shipment );  
console.log( shipmentRainbow instanceof ShipmentRainbow );
```

ECMA SCRIPT 5

# CHAINING PATTERN

# CHAINING PATTERN

```
(function() {
  'use strict';
  // - - - - -
  // define the class
  let _fn = function() {
    this.version = '0.1';
    this.os = 'MacOS X';
    this.browser = 'Chrome';
  };
})()
```

# CHAINING PATTERN

```
_fn.prototype.setVersion = function(version) {  
    this.version = version;  
    return this;  
};  
  
_fn.prototype.setOs = function(os) {  
    this.os = os;  
    return this;  
};  
  
_fn.prototype.setBrowser = function(browser) {  
    this.browser = browser;  
    return this;  
};  
  
_fn.prototype.save = function() {  
    console.log(  
        'saving ' + this.version + ', on ' +  
        this.os + ' with ' + this.browser + '.'  
    );  
    return this;  
};
```

# CHAINING PATTERN

```
Let fn = new _fn();

fn.setVersion('1');
fn.setOs('MacOS XI');
fn.setBrowser('Chrome Xtra');
fn.save();

fn
  .setVersion('2')
  .setOs('MacOS XII')
  .setBrowser('Chrome Xtra Large')
  .save();
```

ECMA SCRIPT 6+

# CLASSES

# KLASSEN UND VERERBUNG

- Definition von Klassen
- Methoden in Klassen
- constructor-Method
- Vererbung mit extend
- Das super-Keyword

# CLASSES

```
class Shape {  
    constructor (id, x, y) {  
        this._id = id  
        this.move(x, y)  
    }  
    move (x, y) {  
        this.x = x  
        this.y = y  
    }  
    get id () {};  
    set id () {};  
}
```

ECMAScript 5 – syntactic sugar: reduced | traditional

```
var Shape = function (id, x, y) {  
    this.id = id;  
    this.move(x, y);  
};  
Shape.prototype.move = function (x, y) {  
    this.x = x;  
    this.y = y;  
};
```

# CLASSES

```
class Rectangle extends Shape {  
    constructor (id, x, y, width, height) {  
        super(id, x, y)  
        this.width = width  
        this.height = height  
    }  
}  
  
class Circle extends Shape {  
    constructor (id, x, y, radius) {  
        super(id, x, y)  
        this.radius = radius  
    }  
}
```

ECMA SCRIPT 6+

# SINGLETON

# SINGLETON

```
class UserStore {  
    constructor(){  
        if(! UserStore.instance){  
            this._data = [];  
            UserStore.instance = this;  
        }  
  
        return UserStore.instance;  
    }  
  
    //rest is the same code as preceding example  
}  
  
const instance = new UserStore();  
Object.freeze(instance);  
  
export default instance;
```