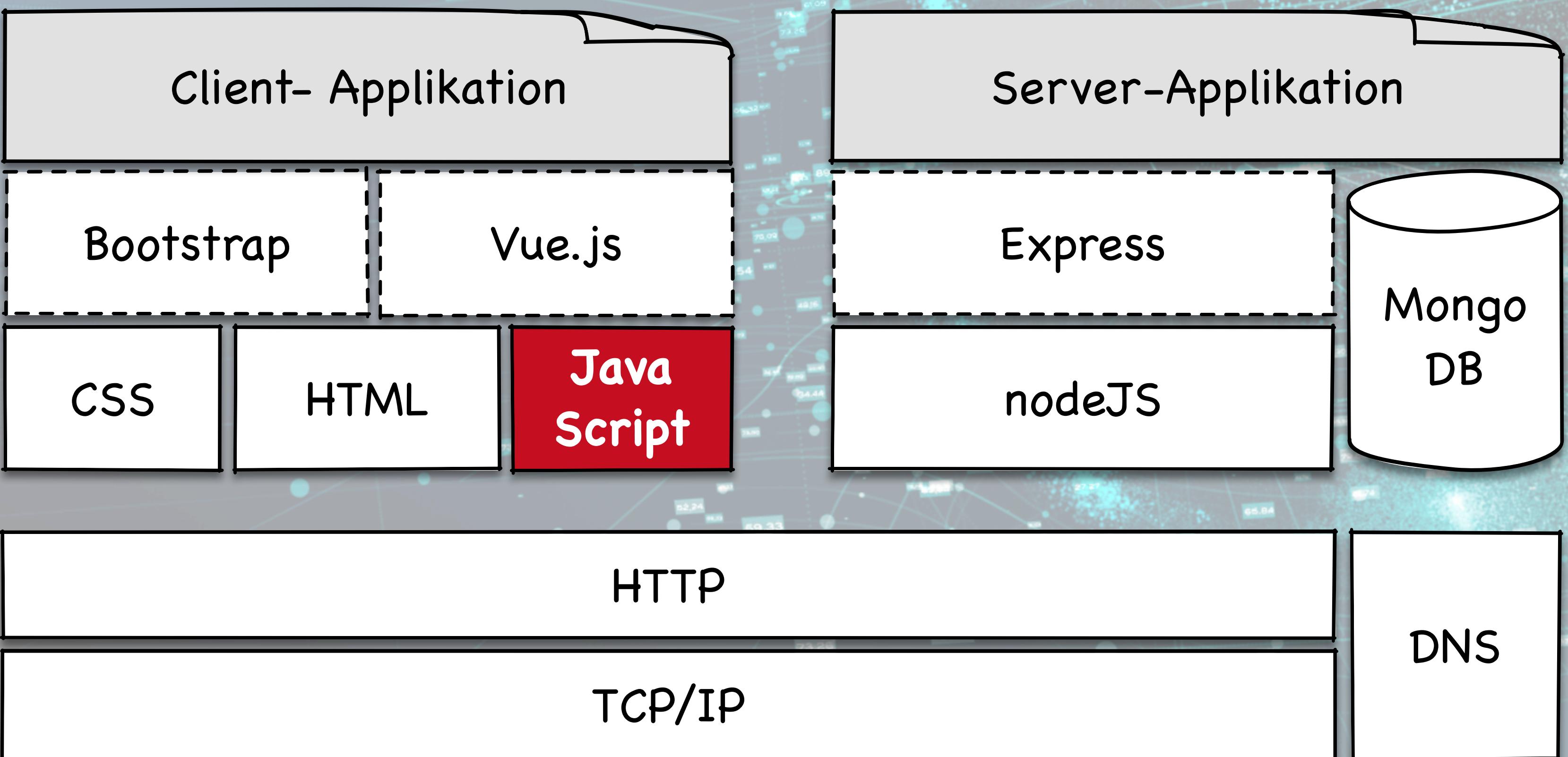


WEB TECHNOLOGIEN 2022

KAPITEL 4: JAVASCRIPT

PROF. DR. AXEL KÜPPER
FACHGEBIET SERVICE-CENTRIC NETWORKING I TU BERLIN & T-LABS

WEBTECHNOLOGIEN ÜBERBLICK



4.1 JAVASCRIPT-EINLEITUNG

GESCHICHTE UND MERKMALE



GESCHICHTE

- Erste Version unter dem Namen *LiveScript* 1995 in zwölf Tagen von Brendan Eich für den *Netscape Navigator* entwickelt
- Namensänderung zu *JavaScript* 1996, basierend auf einer Kooperation zwischen dem Java-Entwickler *Sun* und Netscape
- *ECMAScript*: Standardisierung von JavaScript durch die *European Computer Manufacturer Association (ECMA)*
- Verabschiedung von ECMAScript Version 12 im Juni 2021
- JavaScript ist lediglich eine Implementierung von ECMAScript
- Andere Implementierungen von ECMAScript: *QtScript*, *ActionScript* (Flash) und *ExtendScript* (für Adobe-Produkte)

TYPESCRIPT

- Basiert auf ECMAScript 3 (oder 5) und führt statische Typisierung, Klassen, Vererbung, Module und anonyme Funktionen ein
- Kompiliert zu JavaScript und ist eine syntaktische Obermenge von JS

4.1 JAVASCRIPT-EINLEITUNG

WICHTIGE MERKMALE

SKRIPT-SPRACHEN

- Programmiersprachen, die nicht vor der Ausführung durch einen Compiler übersetzt werden, sondern während der Ausführung durch einen Interpreter
- Einfacher in der Umsetzung als Compiler-Sprachen, da Kompilierungszeit entfällt
- Interpretierte Sprachen benötigen längere Ausführungszeit, da die Übersetzung während der Ausführung erfolgt

DYNAMISCHE TYPISIERUNG

- Datentypen werden dynamisch zur Laufzeit ermittelt
- Keine Möglichkeiten, eine Variable mit einem Typ zu deklarieren
- Typ einer Variablen kann sich zur Laufzeit ändern
- Automatische Konvertierung von Typen, beispielsweise bei Vergleichen mit dem ==-Operator

FUNKTIONALE PROGRAMMIERUNG

- Funktionen als erstklassige Objekte, d.h. sie können Variablen zugewiesen und als Parameter anderer Funktionen verwendet werden
- Deklarativ: Man bestimmt was ein Programm macht, nicht wie es etwas macht

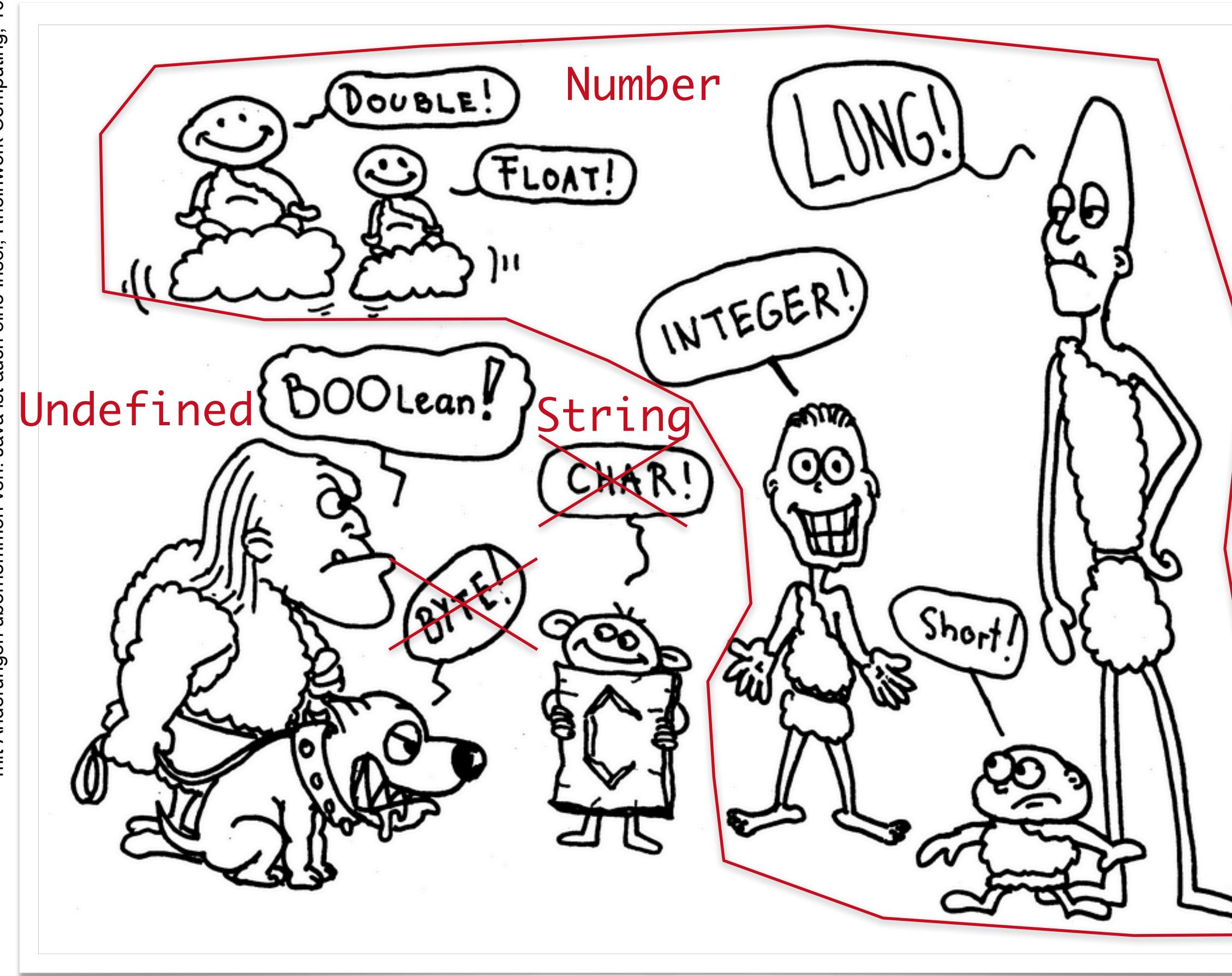
PROTOTYPISCHE OBJEKTORIENTIERUNG

- Umsetzung des objektorientierten Paradigmas basierend auf Prototypen, nicht Klassen

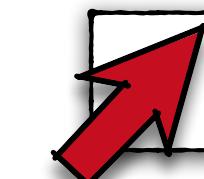
4.1 JAVASCRIPT-EINLEITUNG

DATEN UND DATENTYPEN

IN JAVASCRIPT GIBT ES NUR VIER PRIMITIVE DATENTYPEN



- Daten (Data, Values) werden als Aneinanderreihung von Bits dargestellt, denen ein konkreter Datentyp zugrunde liegt
- Arten von Datentypen
 - Einfache/primitive/elementare Datentypen (Basic Data Types)
 - Komplexe Datentypen (Derived Data Types)
- JavaScript unterstützt vier primitive Datentypen
 - Numbers (Zahlen)
 - Strings (Zeichenketten)
 - Booleans (Wahrheitswerte)
 - Undefined Values (Undefinierte Werte)
- JavaScript unterstützt zwei komplexer Datentypen
 - Objects (Objekte)
 - Functions (Funktionen)



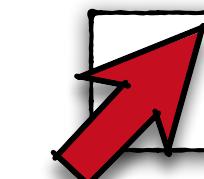
http://eloquentjavascript.net/01_values.html

4.2 PRIMITIVE DATENTYPEN

ZAHLEN

```
> 13
< 13
> 9.81
< 9.81
> 0xffaabbcdd
< 1098081094877
> 01234567
< 342391
> 2.998e8
< 299800000
> 5/0
< Infinity
> 0/0
< NaN
> Infinity-Infinity
< NaN
```

- Keine Unterscheidung zwischen Ganzzahlen und Fließkommazahlen
- Alle Zahlen werden als 64-Bit-Fließkommazahlen dargestellt, d.h. 18 Trillionen verschiedene Zahlen können dargestellt werden
- Schreibweisen
 - Dezimalschreibweise (ohne Präfix)
 - Hexadezimalschreibweise (mit Präfix 0x)
 - Oktalschreibweise (mit Präfix 0)
 - Exponentialschreibweise (mit Infix e)
- Keine Unterstützung der Binärschreibweise
- Spezielle Zahlen
 - **Infinity** und **-Infinity** werden verwendet, wenn ein Wert außerhalb des Wertebereichs liegt
 - **NaN** (Not a Number) wird verwendet, wenn eine Berechnung zu einem Ergebnis führt, welches nicht als Zahl repräsentiert werden kann

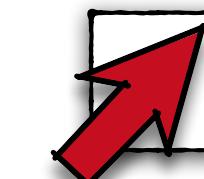


http://eloquentjavascript.net/01_values.html

4.2 PRIMITIVE DATENTYPEN ARITHMETISCHE OPERATIONEN

```
> 100+4*11
< 144
> (100+4)*11
< 1144
> 144%12
< 0
> var zahl=42
< undefined
> zahl++
< 42
> zahl++
< 43
> ++zahl
< 45
> !zahl
< false
> !!zahl
< true
```

Operation	Operator	Beschreibung
Addition	+	Liefert die Summe der Operanden.
Subtraktion	-	Liefert die Differenz.
Multiplikation	*	Liefert das Produkt der Operanden.
Division	/	Liefert den Quotienten.
Modulo	%	Liefert den ganzzahligen Rest der Division der beiden Operanden.
Inkrement	++	Unärer Operator der den Operanden um eins erhöht. Kann sowohl als Präfix- als auch als Postfix-Operator <u>auf Variablen</u> verwendet werden.
Dekrement	--	Unärer Operator der eins vom Operanden subtrahiert. Kann sowohl als Präfix- als auch als Postfix-Operator <u>auf Variablen</u> verwendet werden.
unäre Negation	!	Unärer Operator der die Negation des Operanden liefert. Verwendung als Präfix-Notation



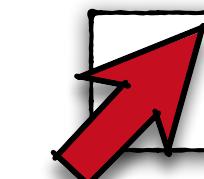
http://eloquentjavascript.net/01_values.html

4.2 PRIMITIVE DATENTYPEN

ZEICHENKETTEN

- Zeichenketten bestehen aus 16-Bit-Zeichen nach USC-2- oder UTF-16-Kodierung und werden durch einfache oder doppelte Anführungszeichen repräsentiert
- Kein Datentyp char zur Darstellung eines einzelnen Zeichens
- Probleme
 - Wie können Steuerzeichen (z.B. zum Einfügen eines Zeilenumbruchs) eingefügt werden?
 - Wie können Zitate in Anführungszeichen Bestandteil einer Zeichenkette sein?
- Backslash \ fungiert als Präfix für Steuerzeichen
 - \n: Zeilenumbruch
 - \t: Tabulator
 - \" : Anführungszeichen als Zeichen in und nicht zum Beenden der Zeichenkette
 - \\: Backslash als Zeichen in der Zeichenkette
- Mehrere Zeichenketten können mit dem + -Operator konkateniert werden

```
> "Eine Schwalbe macht noch keinen Sommer"
< "Eine Schwalbe macht noch keinen Sommer"
> 'Wer zuerst kommt, mahlt zuerst'
< "Wer zuerst kommt, mahlt zuerst"
> "Dies ist die erste Zeile...\n...und dies ist die zweite Zeile"
< "Dies ist die erste Zeile...
 ...und dies ist die zweite Zeile"
> "Ein Zeilenumbruch wird als \"\\n\" geschrieben."
< "Ein Zeilenumbruch wird als "\n" geschrieben."
> "Kon"+"ka"+"te"+"na"+"tion"
< "Konkatenation"
```



http://eloquentjavascript.net/01_values.html

4.2 PRIMITIVE DATENTYPEN

BOOLEAN

```
> 3>2
< true
> "Aachen"<"Zwickau"
< true
> "Peking"!="Beijing"
< true
> NaN==false
< false
> NaN==true
< false
> NaN==NaN
< false

> undefined==undefined
< true
> true&&false
< false
> false||true
< true
> 1+1==2 && 10*10>50
< true
> true ? 1:2
< 1
> false ? 1:2
< 2
```

- Werte vom Typ Boolean können true oder false sein
- Verwendung von Boolean als Ergebnistyp für logische Operatoren und Vergleichsoperatoren (nächste Folie)
- Neben booleschen Werten interpretiert JavaScript auch nicht-boolesche Werte als truthy oder falsy
- undefined, leere Zeichenketten, 0, NaN und null (Literal für optionale Objekte) zählen zu den Werten, die als falsy interpretiert werden, alle anderen Werte als truthy interpretiert

BEACHTE

- Vergleichsoperator == liefert für null==false und null==true in beiden Fällen false
- Vergleichsoperator == liefert für undefined==false und undefined==true in beiden Fällen false
- Vergleichsoperator == liefert für NaN==false und NaN==true in beiden Fällen false
- null und undefined sind nur untereinander gleich
- NaN ist nicht mit sich selber gleich



http://eloquentjavascript.net/01_values.html

4.2 PRIMITIVE DATENTYPEN

VERGLEICHSOPERATOREN UND LOGISCHE OPERATOREN

VERGLEICHSOPERATOREN

Operation	Operator	Beschreibung
Gleichheit	<code>==</code>	Liefert true wenn die Operanden gleich sind.
Ungleichheit	<code>!=</code>	Liefert true wenn die Operanden nicht gleich sind.
strikte Gleichheit	<code>===</code>	Liefert true wenn die Operanden gleich sind und außerdem den gleichen Datentyp haben.
strikte Ungleichheit	<code>!==</code>	Liefert true wenn die Operanden nicht gleich sind und/ oder nicht den gleichen Datentyp haben.
größer als	<code>></code>	Liefert true wenn der linke Operand größer als der rechte ist.
größer oder gleich	<code>>=</code>	Liefert true wenn der linke Operand größer als oder gleich dem rechten Operand ist.
kleiner als	<code><</code>	Liefert true wenn der linke Operand kleiner als der rechte ist.
kleiner oder gleich	<code><=</code>	Liefert true wenn der linke Operand kleiner als oder gleich dem rechten Operand ist.

LOGISCHE OPERATOREN

Operation	Operator	Beschreibung
logisches UND	<code>&&</code>	Binärer Operator der den ersten Operanden zurückgibt falls dieser false ergibt. Ansonsten wird der zweite Operand zurückgegeben.
logisches ODER	<code> </code>	Binärer Operator der den ersten Operanden zurückgibt falls dieser true ergibt. Ansonsten wird der zweite Operand zurückgegeben.
logisches NICHT	<code>!</code>	Unärer Operator den den Operanden negiert
Konditionaler Operator	<code>- ? - : -</code>	Ternärer Operator mit einem Boolean-Ausdruck vor dem Fragezeichen. Bei true wird der Ausdruck links vom Doppelpunkt ausgeführt, bei false der Ausdruck rechts davon

4.2 PRIMITIVE DATENTYPEN

GLEICHHEITSVERGLEICHE

https://developer.mozilla.org/de/docs/Web/JavaScript/comparisons_and_sameness

x	y	==	===	Object.is
undefined	undefined	true	true	true
null	null	true	true	true
true	true	true	true	true
false	false	true	true	true
"foo"	"foo"	true	true	true
{ foo: "bar" }	x	true	true	true
0	0	true	true	true
+0	-0	true	true	false
0	false	true	false	false
""	false	true	false	false
""	0	true	false	false
"0"	0	true	false	false

x	y	==	===	Object.is
"17"	17	true	false	false
[1,2]	"1,2"	true	false	false
new String("foo")	"foo"	true	false	false
null	undefined	true	false	false
null	false	false	false	false
undefined	false	false	false	false
{ foo: "bar" }	{ foo: "bar" }	false	false	false
new String("foo")	new String("foo")	false	false	false
0	null	false	false	false
0	NaN	false	false	false
"foo"	NaN	false	false	false
NaN	NaN	false	false	true

4.2 PRIMITIVE DATENTYPEN

AUTOMATISCHE TYPKONVERTIERUNG

- Wenn Operatoren auf den falschen Datentypen angewendet werden, führt JavaScript im Hintergrund eine automatische Typkonvertierung durch
- Typkonvertierung basiert auf einer großen Anzahl teils komplizierter, teils verwirrender Regeln
- Automatische Typkonvertierung ist in einigen Fällen sinnvoll:
 - Testen ob eine Variable einen richtigen Wert hat oder null bzw. undefined ist
 - Testen ob eine Variable den Wert 0 hat
- In vielen Fällen führt automatische Typkonvertierung zu fehlerhaftem Code und sollte vermieden werden
- Bei Verwendung der strikten Vergleichsoperatoren === und !== wird keine Typkonvertierung durchgeführt

```
> 8*null
< 0
> "5"-1
< 4
> "5"+1
< "51"
> "five"*2
< NaN
> false==0
< true
> null==undefined
< true
> null==0
< false
```

BEACHTE

- undefined ist ein eigener Datentyp und bedeutet, dass einer deklarierten Variablen kein Wert zugewiesen wurde
- null ist ein Objekt, welches einer Variablen zugewiesen werden kann, um zu kennzeichnen, dass die Variable keinen Wert hat



http://eloquentjavascript.net/01_values.html

4.2 PRIMITIVE DATENTYPEN

KONVERTIERUNGSREGELN BEI (LOSER) GLEICHHEIT

https://developer.mozilla.org/de/docs/Web/JavaScript/comparisons_and_sameness

		Operand B						
		Undefined	Null	Number	String	Boolean	Object	
Operand A	Undefined	true	true	false	false	false	IsFalsy(B)	
	Null	true	true	false	false	false	IsFalsy(B)	
	Number	false	false	A === B	A === ToNumber(B)	ToNumber(B) === A	ToPrimitive(B) == A	
	String	false	false	B === ToNumber(A)	A === B	ToNumber(A) === ToNumber(B)	ToPrimitive(B) == A	
	Boolean	false	false	ToNumber(A) === B	ToNumber(A) === ToNumber(B)	A === B	false	
	Object	IsFalsy(A)	IsFalsy(A)	ToPrimitive(A) == B	ToPrimitive(A) == B	false	A === B	

4.3 PROGRAMMSTRUKTUREN AUSDRÜCKE UND ANWEISUNGEN

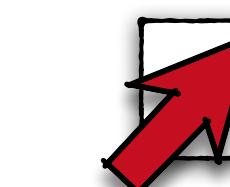
BEISPIELE FÜR AUSDRÜCKE

```
> 13
< 13
> (100+4)*11
< 1144
> 3>2
< true
> "Aachen"<"Zwickau"
< true
```

BEISPIELE ANWEISUNGEN

```
> console.log((100+4)*11);
1144
< undefined
> var stimmung="abendlich";
< undefined
> if (2<5) alert("2<5");
< undefined
> "Aachen"<"Zwickau"
< true
```

- Ein Ausdruck (Expression) ist ein Stück Programmcode, das einen Wert zurück liefert, aber keine Daten modifiziert
- Ausdrücke modifizieren keine Daten
- Ergebnisse von Ausdrücken bilden die Eingabe für andere Ausdrücke, werden in Konstanten oder Variablen gespeichert oder bei Methodenaufrufen übergeben
- Ausdrücke sind oftmals Bestandteil von Anweisungen
- Eine Anweisung (Statement) ist ein Befehl, der den Interpreter veranlasst, etwas zu tun:
 - Ausgabeanweisungen
 - Zuweisungsanweisung
 - Kontrollanweisung
 - Funktionsaufrufe
 - Ausdruck
- Anweisungen haben i.d.R keinen Wert
- Sequentielle Ausführung von Anweisungen ist ein Programm



http://eloquentjavascript.net/02_program_structure.html

4.3 PROGRAMMSTRUKTUREN

VARIABLEN (I)

GETRENNTE DEKLARATION UND INITIALISIERUNG EINER VARIABLEN

```
> var ergebnis;  
< undefined  
> ergebnis=5*5;  
< 25
```

DEKLARATION UND INITIALISIERUNG EINER VARIABLEN IN EINER ANWEISUNG

```
> var ergebnis=5*5;  
< undefined  
> ergebnis;  
< 25
```

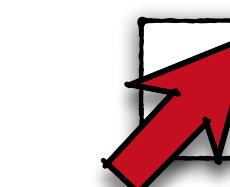
ÜBERSCHREIBEN EINES WERTES EINER VARIABLEN MIT EINEM WERT VOM GLEICHEN DATENTYP

```
> var ergebnis=5*5;  
< undefined  
> ergebnis=6*6;  
< 36
```

ÜBERSCHREIBEN EINES WERTES EINER VARIABLEN MIT EINEM WERT EINES ANDEREN DATENTYPS

```
> var ergebnis=5*5;  
< undefined  
> ergebnis="fünfundzwanzig";  
< "fünfundzwanzig"
```

- Daten werden in Variablen gespeichert und repräsentieren den internen Zustand eines Programmes
- Deklaration einer Variablen erfolgt mit dem Schlüsselwort `var` gefolgt vom Namen der Variablen
- Optional kann die Variable während der Deklaration durch den `=`-Operator gefolgt von einem Ausdruck mit einem Wert initialisiert werden
- Typinferenz (Type Inference): Deklaration einer Variablen erfolgt ohne Typangabe, d.h. Bestimmung des Datentyps erfolgt automatisch zur Laufzeit
- Nachdem eine Variable definiert wurde, kann sie als Ausdruck oder in einem Ausdruck verwendet werden
- Variablen können während der Laufzeit des Programms verschiedene Werte unterschiedlicher Datentypen annehmen
- Variablen primitiver Datentypen speichern den Wert, Variablen komplexer Datentypen speichern die Referenz auf den Wert



4.3 PROGRAMMSTRUKTUREN VARIABLEN (II)

```
> Label: {
    console.log("Eins");
    console.log("Zwei");
    console.log("Drei");
}
Eins
Zwei
Drei
< undefined

> var x=0;
< undefined
> while (x<10) {
    x++
}
< 9

> var x=1;
{
    var x=2;
}
console.log(x);
2
< undefined
```

- Eine Blockanweisung gruppiert 0 oder mehrere Anweisungen
- Ein Block wird durch ein Paar geschweifte Klammern abgegrenzt und kann optional mit einem Label gekennzeichnet werden
- Eine Blockanweisungen wird meistens in Verbindung mit Kontrollflussanweisungen (if/else, for, while) genutzt
- Unter dem Block Scope (Block-Sichtbarkeitsbereich) einer Variablen versteht man den Programmabschnitt, in dem eine Variable sicht- und nutzbar ist
- Variablen, die mit var deklariert werden, haben keinen Block Scope - sie sind an den Function Scope der umschließenden Funktion oder des Skripts gebunden (siehe Folien 26 und 27)

4.3 PROGRAMMSTRUKTUREN VARIABLEN (II)

```
> let a;  
< undefined  
> let name="Simon";  
< undefined
```

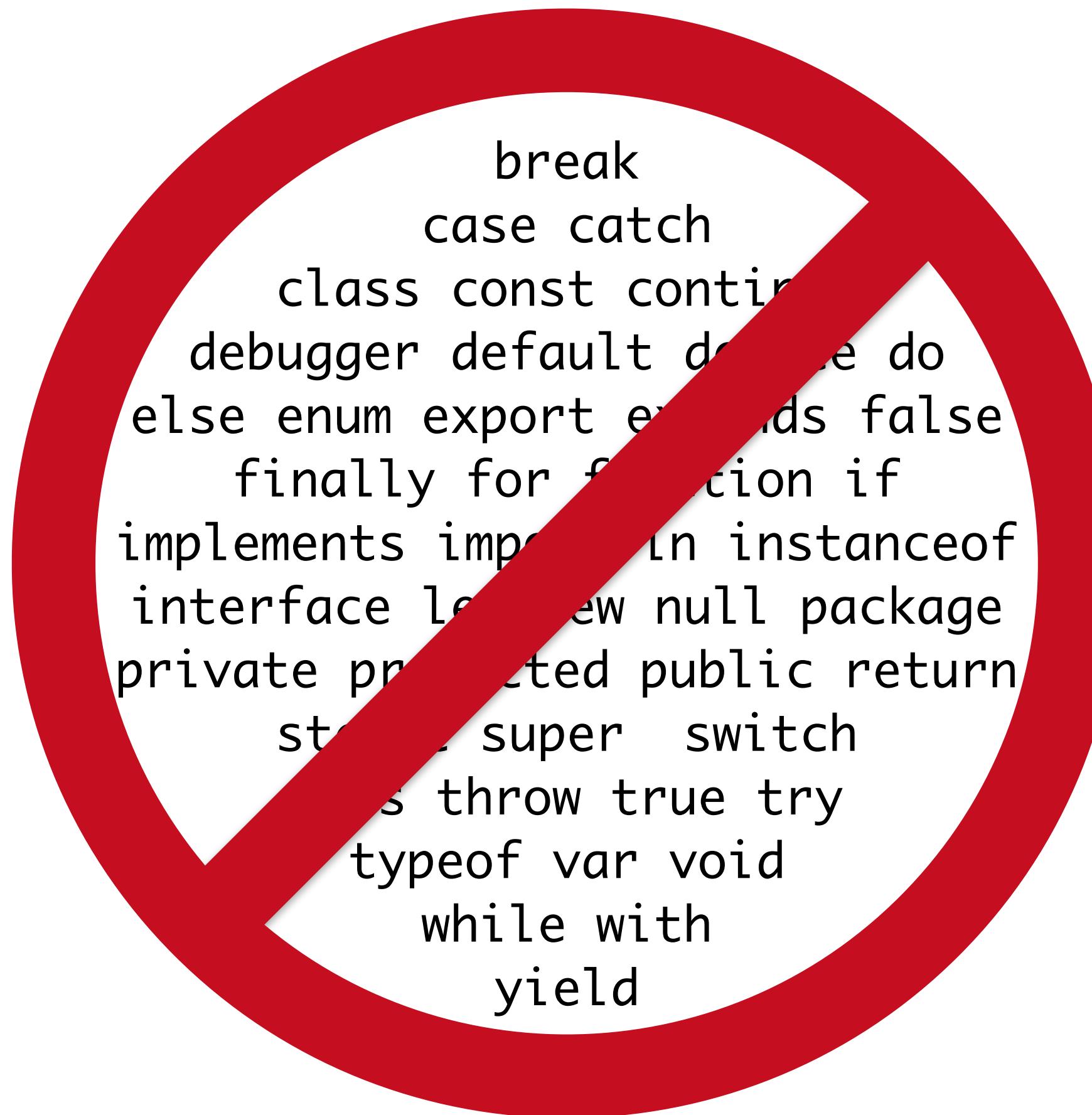
```
> {  
    let y=2;  
}  
console.log(y);  
✖ > Uncaught ReferenceError: y is not defined  
      at <anonymous>:4:13
```

```
> const Pi=3.14;  
< undefined  
> Pi=1;  
✖ > Uncaught TypeError: Assignment to constant variable.  
      at <anonymous>:1:3
```

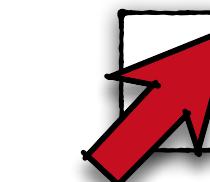
- Seit Version 6 von ECMAScript können Variablen anstelle von `var` auch mit den Schlüsselwörtern `let` und `const` deklariert werden
- Mit `let` können Variablen auf Blockebene deklariert werden, d.h. der Scope der Variablen wird durch den umgebenden Block gebildet
- Außerhalb des umgebenden Blockes ist eine mit `let` deklarierte Variable nicht zugreifbar
- Mit `const` können nicht veränderbare Variablen (Konstanten) deklariert werden, deren Scope durch den umgebenden Block gegeben ist

4.3 PROGRAMMSTRUKTUREN

SCHLÜSSELWÖRTER - VERBOTEN ALS VARIABLENNAMEN

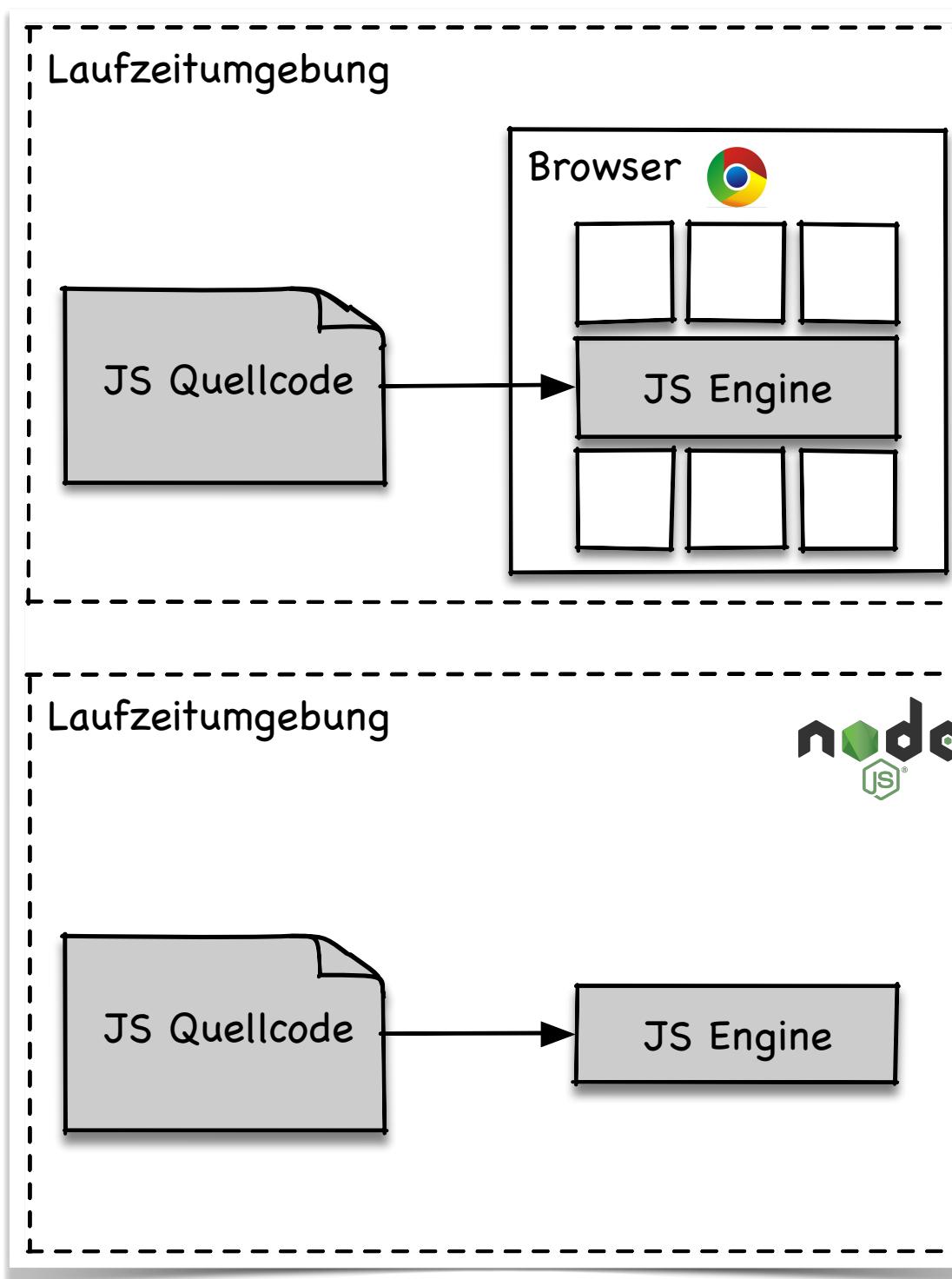


- Variablennamen müssen mit einem Buchstaben, einem Unterstrich oder dem Dollarzeichen beginnen, gefolgt von Buchstaben, Ziffern oder dem Unterstrich
- Anweisungen setzen sich aus Schlüsselwörtern zusammen
- Schlüsselwörter dürfen nicht als Variablennamen verwendet werden



http://eloquentjavascript.net/02_program_structure.html

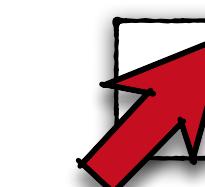
4.3 PROGRAMMSTRUKTUREN JAVASCRIPT-UMGEBUNGEN



- Menge aller Variablen, die zu einer gegebenen Zeit existieren, wird als Umgebung oder Laufzeitumgebung (Environment) bezeichnet
- Laufzeitumgebung ist zum Start eines Programms nicht leer, sondern enthält bereits deklarierte und initialisierte Variablen, die Bestandteil des Sprachstandards ECMAScript oder des umgebenden Systems sind

JAVASCRIPT-UMGEBUNGEN IN DER VORLESUNG

- Webbrowser: enthält Variablen und Funktionen, um Inhalte einer Webseite zu lesen und zu verändern oder um Nutzerinteraktionen zu erkennen, siehe Document Object Model (DOM)
- node.js: enthält Module mit Variablen und Funktionen um ein ausführbares Programm, z.B. einen Webserver, zu erzeugen



http://eloquentjavascript.net/02_program_structure.html

4.3 PROGRAMMSTRUKTUREN KONTROLLFLÜSSE (I)

- Als Kontrollfluss eines Programms bezeichnet man die Abarbeitung von Anweisungen in einer bestimmten Reihenfolge

SEQUENTIELLER KONTROLLFLUSS

- Bei einem sequentiellen Kontrollfluss werden alle Anweisungen genau einmal in der Reihenfolge ihres Auftretens im betrachteten Programm(teil) ausgeführt

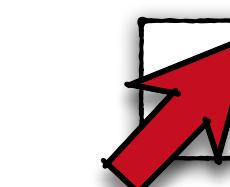
BEDINGTE AUSFÜHRUNG

- Bei der bedingten Ausführung werden bestimmte Anweisungen nur bei Erfüllung einer Bedingung ausgeführt
- Mit der `if`-Anweisung wird eine Sequenz von Anweisungen ausgeführt oder ausgelassen, je nachdem ob der folgende boolesche Ausdruck wahr oder falsch ist
- Mit dem Schlüsselwort `else` wird eine alternative Sequenz von Anweisungen ausgeführt wenn der boolesche Ausdruck unwahr ist

```
> var zahl=Number(prompt("Wähle eine Zahl", ""));
  alert("Deine Zahl ist die Wurzel von "+zahl*zahl);
< undefined

> var zahl=Number(prompt("Wähle eine Zahl", ""));
  if (!isNaN(zahl))
    alert("Deine Zahl ist die Wurzel von "+zahl*zahl);
< undefined

> var zahl=Number(prompt("Wähle eine Zahl", ""));
  if (!isNaN(zahl))
    alert("Deine Zahl ist die Wurzel von "+zahl*zahl);
  else
    alert("Hey, warum hast mir keine Zahl genannt?");
< undefined
```



http://eloquentjavascript.net/02_program_structure.html

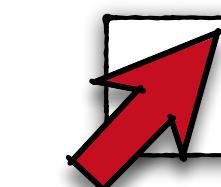
4.3 PROGRAMMSTRUKTUREN KONTROLLFLÜSSE (II)

BEDINGTE AUSFÜHRUNG MIT SWITCH

- Mehrere if/else-Anweisungen können miteinander verknüpft werden
- switch-Anweisung ist oftmals anstelle von langen Ketten von if/else-Anweisungen vorzuziehen
- Ein case ist ein Einsprungpunkt, ab dem mit der Ausführung einer Sequenz von Anweisungen begonnen wird
- Ausdruck hinter switch wird evaluiert und mit den Werten hinter case verglichen
- Bei Übereinstimmung wird mit der Ausführung ab der Anweisung hinter case fortgesetzt bis zum Ende der switch-Blockes
- Bei einem break wird die Ausführung abgebrochen und der switch-Block verlassen
- Gibt es für keinen case-Wert eine Übereinstimmung, wird mit den Anweisungen hinter default fortgesetzt

```
> var wetter=prompt("Wie ist das Wetter heute?");  
if (wetter=="regnerisch") console.log("Denk an den Regenschirm");  
else if (wetter=="sonnig") console.log("Denk an die Sonnenbrille");  
else if (wetter=="bewölkt") console.log("Geh vor die Türe");  
else console.log("Unbekannte Wetterart");  
Denk an die Sonnenbrille  
< undefined
```

```
> switch(prompt("Wie ist das Wetter heute?")) {  
  case "regnerisch": console.log("Denk an den Regenschirm");  
    break;  
  case "sonnig": console.log("Denk an die Sonnenbrille");  
    break;  
  case "bewölkt": console.log("Geh vor die Türe");  
    break;  
  default: console.log("Unbekannte Wetterart");  
}  
Denk an die Sonnenbrille  
< undefined
```



4.3 PROGRAMMSTRUKTUREN KONTROLLFLÜSSE (III)

```
> var zahl=0;
while(zahl<=12) {
    console.log(zahl);
    zahl=zahl+2;
}
0
2
4
6
8
10
12
< 14
```

```
> do {
    var deinName=prompt("Wer bist Du?");
} while (!deinName);
console.log(deinName);
Axel
< undefined
```

SCHLEIFEN

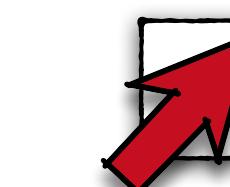
- Schleifen ermöglichen die wiederholte Ausführung einer bestimmten Sequenz von Anweisungen basierend auf einer Bedingung

SCHLEIFEN MIT WHILE

- Bei while steht eine *Ausführungsbedingung* als boolescher Ausdruck am Beginn der Schleife
- Die Sequenz von Anweisungen wird ausgeführt, wenn die Ausführungsbedingung wahr ist, d.h. sie wird einmal, einmal oder mehrmals ausgeführt

SCHLEIFEN MIT DO

- Bei do steht die Ausführungsbedingung am Ende der Schleife, d.h. die Schleife wird mindestens einmal ausgeführt



http://eloquentjavascript.net/02_program_structure.html

4.3 PROGRAMMSTRUKTUREN KONTROLLFLÜSSE (IV)

```
> var ergebnis=1;
  for (var zaehler=0; zaehler<10; zaehler=zaehler+1)
    ergebnis=ergebnis*2;
  console.log(ergebnis);
1024
<- undefined

> for (var zahl=20; ; zahl++) {
  if (zahl%7==0)
    break;
  console.log(zahl);
21
<- undefined
```

SCHLEIFEN MIT FOR

- for-Schleifen sind eine kompaktere Form von Schleifen mit drei Anweisungen im Kopf der Schleife
 - die erste Anweisung initialisiert die Schleife
 - die zweite Anweisung enthält die Ausführungsbedingung als booleschen Ausdruck
 - die dritte Anweisung aktualisiert den Zustand der Schleife, üblicherweise die Zählvariable
- Neben einer zu `false` abgeleiteten Ausführungsbedingung können for-Schleifen mit `break` abgebrochen werden
- Mit `continue` wird die gegenwärtige Iteration der for-Schleife beendet und die nächste Iteration begonnen



http://eloquentjavascript.net/02_program_structure.html

4.4 FUNKTIONEN

FUNKTIONSAUSDRÜCKE UND FUNKTIONSANWEISUNGEN

FUNKTIONSAUSDRUCK

```
> var addition=function additionsFunktion(zahl1, zahl2) {  
    return zahl1+zahl2;  
};  
< undefined  
> addition.name;  
< "additionsFunktion"  
> additionsFunktion(1,2);  
✖ > Uncaught ReferenceError: additionsFunktion is not defined  
      at <anonymous>:1:1  
> addition(1,2);  
< 3
```

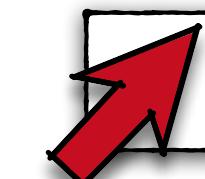
ANONYMER FUNKTIONSAUSDRUCK

```
> var addition=function(zahl1, zahl2) {  
    return zahl1+zahl2  
};  
< undefined  
> addition.name;  
< "addition"  
> addition(1,2);  
< 3
```

FUNKTIONSANWEISUNG

```
> function addition(zahl1, zahl2) {  
    return zahl1+zahl2;  
};  
< undefined  
> addition.name;  
< "addition"  
> addition(1,2);  
< 3
```

- Funktionen kapseln Programmcode und dienen der Strukturierung von Programmen, der Wiederverwendung, der Verknüpfung von Programmcode mit intuitiven Namen und der Isolation von unterschiedlichen Teilen eines Programmes
- Funktionen können als Funktionsausdruck oder Funktionsanweisung erzeugt werden
- Ein Funktionsausdruck weist einer Variablen als Wert eine Funktion zu
- Neben der Variablenzuweisung kann die Funktion einen eigenen Namen haben oder als anonyme Funktion definiert werden
- Funktionsaufruf erfolgt über den Namen der Variablen
- Eine Funktionsanweisung erzeugt eine Funktion eines bestimmten Namens
- Funktionsaufruf erfolgt über den Namen der Funktion bzw. der Variablen gefolgt von den Argumenten in Klammern
- Der Aufruf einer Funktion ohne Argumente muss mit leeren Klammern erfolgen



4.4 FUNKTIONEN

STRUKTUR VON FUNKTIONEN

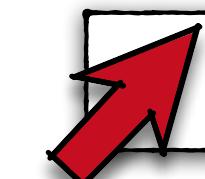
EINE FUNKTION OHNE SEITENEFFEKTE UND MIT DYNAMISCHER TYPERKENNTUNG

```
> var addition=function(zahl1, zahl2) {  
    return zahl1+zahl2;  
};  
< undefined  
> addition(1,2);  
< 3  
> addition("Web", "technologien");  
< "Webtechnologien"
```

EINE FUNKTION MIT SEITENEFFEKTEN

```
> var makeNoise=function() {  
    console.log("Pling!");  
};  
< undefined  
> makeNoise;  
< f () {  
    console.log("Pling!");  
}  
> makeNoise();  
Pling!  
< undefined
```

- Funktionsdefinitionen bestehen aus einem
 - Funktionskopf zur Benennung der Übergabeparameter
 - Funktionsrumpf, der die Programmlogik enthält
- Funktionsrumpf ist immer in geschweiften Klammern eingebettet, auch wenn er nur aus einer einzelnen Zeile besteht
- Eine Funktion kann keine, einen oder mehrere Übergabeparameter oder Argumente haben
- Eine Funktion hat optional einen Rückgabeparameter
- Der Rückgabeparameter wird durch das Schlüsselwort `return` gefolgt vom Ausdruck, der den Rückgabewert repräsentiert, gekennzeichnet
- Eine Funktion ohne Rückgabeparameter erzeugt i.d.R. Seiteneffekte (z.B. Ausgabe auf dem Bildschirm)
- Eine Funktion mit Rückgabeparameter kann ebenfalls Seiteneffekte erzeugen
- Datentypen der Übergabe- und Rückgabeparameter werden dynamisch zur Laufzeit ermittelt



http://eloquentjavascript.net/03_functions.html

4.4 FUNKTIONEN

SCOPES VON FUNKTIONEN

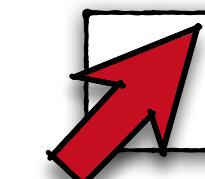
FUNKTION MIT EINER LOKALEN VARIABLEN

```
> var power=function(base, exponent) {  
    var result=1;  
    for (var count=0; count<exponent; count++)  
        result*=base;  
    return result;  
};  
< undefined  
> power(2,10);  
< 1024
```

GLOBALE UND LOKALE VARIABLEN MIT DEM GLEICHEN NAMEN

```
> var x="outside";  
var f1=function() {  
    var x="inside f1";  
};  
f1();  
x;  
< "outside"  
> var x="outside";  
var f2=function() {  
    x="inside f2";  
};  
f2();  
x;  
< "inside f2"
```

- Unter dem Scope (Sichtbarkeitsbereich) einer Variablen versteht man den Programmabschnitt, in dem eine Variable sicht- und nutzbar ist
- Beachte Unterschied zwischen Block Scope (Kapitel 4.3) und Function Scope
- Übergabeparameter und Variablen, die innerhalb einer Funktion deklariert werden, sind lokal und nicht außerhalb der Funktion sichtbar, d.h. der Function Scope ist die umgebende Funktion
- Variablen, die außerhalb aller Funktionen deklariert werden, sind global und überall im Programm sichtbar
- Auf globale Variablen kann innerhalb von Funktionen zugegriffen werden, wenn die jeweilige Funktion nicht eine Variable mit dem gleichen Namen deklariert



http://eloquentjavascript.net/03_functions.html

4.4 FUNKTIONEN NESTED SCOPES

```
> var landscape = function() {
    var result = "";
    var flat = function(size) {
        for (var count = 0; count < size; count++)
            result += "_";
    };
    var mountain = function(size) {
        result += "/";
        for (var count = 0; count < size; count++)
            result += "=\"";
        result += "\\\"";
    };
    flat(3);
    mountain(4);
    flat(6);
    mountain(1);
    flat(1);
    return result;
};
< undefined
> landscape();
< "___/\"\"\"\\____/\"\""
```

```
> var something = 1;
{
    var something = 2;
    console.log(something);
}
console.log(something);
2
2
< undefined
```

- Nested Scopes sind verschiedene Ebenen von Sichtbarkeitsbereichen von Variablen, die entstehen, wenn Funktionen in andere Funktionen eingebettet werden
- Beispiel: `result` kann in den Funktionen `flat` und `mountain` gesehen und genutzt werden, da sie in der umgebenden Funktion deklariert wird
- `flat` und `mountain` können nicht gegenseitig ihre `count`-Variable sehen, da sie außerhalb des Scope der jeweils anderen Funktion liegen
- Wiederholung: Variablen, die mit `var` deklariert werden, haben keinen Block Scope - sie sind an den Function Scope der umschließenden Funktion oder des Skripts gebunden



http://eloquentjavascript.net/03_functions.html

4.4 FUNKTIONEN

POSITION VON FUNKTIONSDEKLARATIONEN IM PROGRAMM

```
> console.log("The future says:", future());
  function future() {
    return "We STILL have no flying cars.";
}
The future says: We STILL have no flying cars.
```

```
> console.log("The future says:", future());
  var future=function() {
    return "We STILL have no flying cars.";
}
✖ > Uncaught TypeError: future is not a function
      at <anonymous>:1:33
```

```
> function example() {
  function a() {}
  if (something) {
    function b() {}
  }
}
undefined
```

- Funktionsanweisungen können überall im Programm eingefügt werden, auch dann, wenn die Funktion vor der Funktionsanweisung aufgerufen wird
- Hoisting (Heben): intern verschiebt der Interpreter alle Funktionsdeklarationen vor der Ausführung eines Scope an seinen Beginn
- Hoisting funktioniert nicht für Funktionsausdrücke

- Funktionsanweisungen und -ausdrücke innerhalb eines bedingten Blockes einer `if`-Anweisung sind erlaubt, aber problematisch und sollten daher vermieden werden



http://eloquentjavascript.net/03_functions.html

4.4 FUNKTIONEN

OPTIONALE ARGUMENTE

```
> function power(base, exponent) {  
  if (exponent == undefined)  
    exponent = 2;  
  var result = 1;  
  for (var count = 0; count < exponent; count++)  
    result *= base;  
  return result;  
};  
< undefined  
> power(4);  
< 16  
> power(4,3);  
< 64
```

- JavaScript ist sehr tolerant hinsichtlich der Anzahl der Parameter, die einer Funktion beim Aufruf übergeben werden
 - Wenn beim Aufruf mehr Werte übergeben werden, als die Funktion Parameter definiert, werden die restlichen ignoriert
 - Wenn beim Aufruf weniger Werte übergeben werden, als die Funktion Parameter definiert, werden die fehlenden Parameter mit `undefined` belegt
- Verhalten ermöglicht die Definition von Funktionen mit optionalen Parametern und von solchen, die eine beliebige Anzahl von Parametern akzeptieren und bearbeiten können



http://eloquentjavascript.net/03_functions.html

4.5 FELDER UND OBJEKTE

FELDER

```
> var obst=[];
< undefined
> obst[0]="Apfel";
< "Apfel"
> obst[1]="Kirsche";
< "Kirsche"
> obst[2]="Birne";
< "Birne"
> obst
< ▶ (3) ["Apfel", "Kirsche", "Birne"]
> var gemuese=["Paprika", "Tomate", "Gurke"];
< undefined
> gemuese
< ▶ (3) ["Paprika", "Tomate", "Gurke"]

> var buch=new Array("Akio Morita", "Made in Japan", "3-423-02336-7", "199
< undefined
> buch
< ▶ (4) ["Akio Morita", "Made in Japan", "3-423-02336-7", "1994"]
```

- Ein Feld (Array) ist ein zusammengesetzter Datentyp, der mehrere Werte unter einem gemeinsamen Namen speichert und die Werte über einen Index anspricht
- Jedes Element eines Feldes kann von jedem Datentyp sein - Integer, String, Boolean, ein Feld oder ein Objekt
- Felder können mit der Literal-Notation oder mit einem Array-Konstruktor erzeugt werden



http://eloquentjavascript.net/04_data.html

4.5 FELDER UND OBJEKTE

MEHRDIMENSIONALE FELDER

```
> var buecher=new Array();
< undefined
> buecher[0] = new Array("Eine kurze Geschichte","Bryson, Bill", 2003);
< ▶ (3) ["Eine kurze Geschichte", "Bryson, Bill", 2003]
> buecher[1] = new Array("Accidental Empires","Cringely, Robert X.", 1992);
< ▶ (3) ["Accidental Empires", "Cringely, Robert X.", 1992]
> buecher[2] = new Array("Arm und Reich","Diamond, Jared", 1997);
< ▶ (3) ["Arm und Reich", "Diamond, Jared", 1997]
> buecher
< ▶ (3) [Array(3), Array(3), Array(3)] ⓘ
  ▶ 0: (3) ["Eine kurze Geschichte", "Bryson, Bill", 2003]
  ▶ 1: (3) ["Accidental Empires", "Cringely, Robert X.", 1992]
  ▶ 2: (3) ["Arm und Reich", "Diamond, Jared", 1997]
    length: 3
  ▶ __proto__: Array(0)
> buecher[1][2];
< 1992
```

- Felder können mehrdimensional sein, d.h. die Elemente eines Feldes dürfen selber wieder Felder sein, welche wiederum Elemente mit unterschiedlichen Typen enthalten

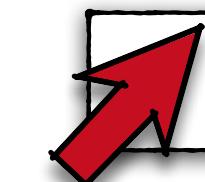
4.5 FELDER UND OBJEKTE EIGENSCHAFTEN

```
> var listOfNumbers=[2,3,5,7,11];
< undefined
> listOfNumbers.length;
< 5
> var name="length";
< undefined
> listOfNumbers[name];
< 5
> listOfNumbers["length"];
< 5
> var i=3;
< undefined
> listOfNumbers[i];
< 7
```

Eigenschaften bei mehrdimensionalen Feldern

```
> bucher[1];
< ▶ (3) ["Accidental Empires", "Cringely, Robert X.", 1992]
> bucher[1][0];
< "Accidental Empires"
```

- Fast alle JavaScript-Werte haben Eigenschaften
- myString.length ist die length-Eigenschaft von myString
- Math.max ist die max-Eigenschaft von Math
- Ausnahmen: null und undefined haben keine Eigenschaften
- Eigenschaften können auf zwei Arten angesprochen
 - dot-Notation, z.B. value.x: der Teil nach dem Punkt muss ein gültiger Bezeichner sein, welcher die Eigenschaft benennt
 - Klammer-Notation, z.B. value[x]: innerhalb der eckigen Klammern befindet sich ein Ausdruck, der evaluiert wird, um den Namen der Eigenschaft zu erhalten
- Die Elemente eines Feldes sind Eigenschaften
- Die Namen dieser Eigenschaften sind Zahlen, die oftmals in Variablen gespeichert werden
- Daher werden die Elemente eines Feldes durch die Klammer-Notation angesprochen



http://eloquentjavascript.net/04_data.html

4.5 FELDER UND OBJEKTE

METHODEN

Methoden für Arrays

Methode	Funktion
concat()	Hängt Elemente oder Felder an ein bestehendes Feld an.
filter()	Filtert Elemente aus dem Feld auf Basis eines in Form einer Funktion übergebenen Filterkriteriums.
forEach()	Wendet eine übergebene Funktion auf jedes Element im Feld an.
join()	Wandelt ein Feld in eine Zeichenkette um.
map()	Bildet die Elemente eines Feldes auf Basis einer übergebenen Umwandlungsfunktion auf neue Elemente ab.
pop()	Entfernt das letzte Element eines Feldes.
push()	Fügt ein neues Element am Ende des Feldes ein.
reduce()	Fasst die Elemente eines Feldes auf der Basis einer übergebenen Funktion zu einem Wert zusammen.
reverse()	Kehrt die Reihenfolge der Elemente im Feld um.
shift()	Entfernt das erste Element eines Feldes.
slice()	Schneidet einzelne Elemente aus einem Feld heraus.
splice()	Fügt neue Elemente an beliebiger Position im Feld hinzu.
sort()	Sortiert das Feld, optional auf Basis einer übergebenen Vergleichsfunktion.

- Eigenschaften, die auf Funktionen verweisen, werden als Methoden bezeichnet

```
> var slogan=[];
<- undefined
> slogan.push("Wir");
<- 1
> slogan.push("haben", "die", "Ideen", "für", "Zukunft");
<- 6
> slogan
<- ▶ (6) ["Wir", "haben", "die", "Ideen", "für", "Zukunft"]
> slogan.pop();
<- "Zukunft"
> slogan.push("morgen");
<- 6
> slogan
<- ▶ (6) ["Wir", "haben", "die", "Ideen", "für", "morgen"]
```

4.5 FELDER UND OBJEKTE

OBJEKTE (I)

```
> var vorlesung = {
  title: "Webtechnologien",
  description: "Vorlesung für *informatiker",
  places: 18,
  readPlaces: function() {
    console.log("Anzahl der Plätze für "+this.title+": "+this.places);
  },
  writePlaces: function(newNumber) {
    this.places=newNumber;
  }
};
< undefined
> vorlesung.title;
< "Webtechnologien"
> vorlesung.readPlaces();
  Anzahl der Plätze für Webtechnologien: 18
< undefined
> vorlesung.writePlaces(300);
< undefined
> vorlesung.readPlaces();
  Anzahl der Plätze für Webtechnologien: 300
< undefined
> vorlesung.raum="H2020";
< "H2020"
> vorlesung.raum;
< "H2020"
```

- Objekte sind beliebige Kollektionen von Eigenschaften. Objekte werden durch geschweifte Klammern zusammen gehalten
- Eigenschaften in der Key/Value-Schreibweise: d.h. Name der Eigenschaft, Doppelpunkt, Werte
- Wert einer Eigenschaft kann ein Literal, eine Variable eines primitiven Datenobjekts, ein Feldes, ein Objekt oder eine Funktion sein
- Eigenschaften werden durch Komma getrennt
- Namen von Eigenschaften müssen gültige Bezeichner sein, die Groß- und Kleinbuchstaben, Zahlen, \$ und _ enthalten und nicht mit einer Zahl beginnen
- Namen, die hiervon abweichen, müssen in Anführungszeichen geschrieben werden
- Neue Elemente können einem Objekt hinzugefügt werden durch Zuweisung eines Wertes an eine Eigenschaft mit einem noch nicht vergebenen Namen



http://eloquentjavascript.net/04_data.html

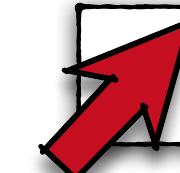
4.5 FELDER UND OBJEKTE

OBJEKTE (II)

```
> vorlesung.raum="H2020";
< "H2020"
> vorlesung.raum;
< "H2020"
> delete vorlesung.raum;
< true
> vorlesung.raum;
< undefined
> "raum" in vorlesung;
< false
> "places" in vorlesung;
< true
```

```
> var list0fNumbers=[2,3,5,7,11];
< undefined
> typeof list0fNumbers;
< "object"
```

- Eigenschaften werden aus einem Objekt mit Hilfe des `delete`-Operators gelöscht
- Der `in`-Operator prüft, ob ein Objekt eine Eigenschaft mit dem in Anführungszeichen angegeben Namen besitzt
- Merke: Felder sind eigentlich spezielle Objekte, die dafür ausgelegt sind, Sequenzen von Daten zu speichern
- Es existiert kein Datentyp `Array` in JavaScript



http://eloquentjavascript.net/04_data.html

4.5 FELDER UND OBJEKTE

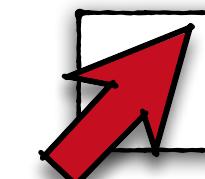
PRIMITIVE DATENTYPEN VS OBJEKTE

```
> var wert1=10;  
< undefined  
> var wert2=wert1;  
< undefined  
> wert1==wert2;  
< true  
> wert1=15;  
< 15  
> wert1==wert2;  
< false  
> wert2;  
< 10
```

```
> var object1={wert: 10};  
< undefined  
> var object2=object1;  
< undefined  
> object1==object2;  
< true  
> object1.wert=15;  
< 15  
> object1==object2;  
< true  
> object2.wert;  
< 15
```

- Variablen primitiver Datentypen speichern den konkreten Wert
- Der `==`-Operator liefert `true`, wenn es sich um denselben Wert handelt
- Werte von primitiven Datentypen sind unveränderbar (immutable) - es ist nicht möglich den existierenden Wert eines primitiven Datentyps nachträglich zu verändern

- Objektvariablen speichern den Verweis auf ein Objekt
- Der `==`-Operator liefert `true`, wenn es sich um dieselben Objekte handelt, nicht um die gleichen
- Objektwerte sind veränderbar (mutable) - es ist möglich ihre Eigenschaften nachträglich zu verändern



4.5 FELDER UND OBJEKTE

CALL-BY-VALUE VS CALL-BY-REFERENCE

```
> var test=function(wertParameter) {  
    wertParameter=15;  
};  
< undefined  
> var testWert=10;  
< undefined  
> test(testWert);  
< undefined  
> testWert;  
< 10
```

```
> var test=function(objectParameter) {  
    objectParameter.wert=15;  
};  
< undefined  
> var testObject={wert: 10};  
< undefined  
> test(testObject);  
< undefined  
> testObject.wert;  
< 15
```

- Wertparameter (call-by-value) sind Parameter von Funktionen, die beim Aufruf eine Kopie der übergebenen Argumente speichern und im inneren der Funktion auf dieser Kopie arbeiten
- Werte primitiver Datentypen werden in JavaScript als Wertparameter behandelt

- Referenzparameter (call-by-reference) sind Parameter von Funktionen, die beim Aufruf eine Referenz auf das übergebene Argumente speichern und im inneren der Funktion mit dieser Referenz arbeiten
- Objektwerte werden in JavaScript als Referenzparameter behandelt

4.5 FELDER UND OBJEKTE KONSTRUKTOREN (I)

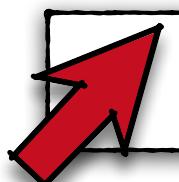
```
> function Vorlesung(title, description, places) {
  this.title=title;
  this.description=description;
  this.places=places;
  this.readPlaces=function() {
    console.log("Anzahl der Plätze: "+this.places);
  }
  this.writePlaces=function(newNumber) {
    this.places=newNumber;
  }
};
< undefined
> var webtech=new Vorlesung("Webtechnologien", "Vorlesung für *informatiker", 300);
< undefined
> webtech;
< ▶ Vorlesung {title: "Webtechnologien", description: "Vorlesung für *info rmatiker", places: 300, readPlaces: f, writePlaces: f}
> webtech.readPlaces();
  Anzahl der Plätze: 300
< undefined
```

- Manchmal benötigt man mehrere Objekte mit denselben Eigenschaften
- ES 6 (JS 2015) erlaubt die Definition von Klassen und deren Instanziierung als Objekte
- Alternative: Definition von Konstruktoren
- Ein Konstruktor (constructor) ist eine Funktion, die ein Objekt mit bestimmten Eigenschaften erstellt
- Ein Konstruktor wird aufgerufen mit dem Schlüsselwort `new` gefolgt vom Namen des Konstruktors und den Werten der Eigenschaften als Aufrufparameter
- Konvention: Konstruktorfunktionen sollten mit einem Großbuchstaben beginnen

4.5 FELDER UND OBJEKTE KONSTRUKTOREN (II)

```
> var x1=new Object();
< undefined
> var x2=new String();
< undefined
> var x3=new Number();
< undefined
> var x4=new Boolean();
< undefined
> var x5=new Array();
< undefined
> var x6=new RegExp();
< undefined
> var x7=new Date();
< undefined
```

- JavaScript hat eine Reihe von "eingebauten" Konstruktoren zur Erzeugung bestimmter Objekte
- Diese Konstruktoren enthalten eine Reihe nützlicher Methoden zur Bearbeitung der Daten
- String, Number und Boolean erzeugen keine primitiven Datentypen sondern komplexe Objekte, deren Bearbeitung bei rechenintensiven Operationen viel langsamer ist, als die Bearbeitung primitiver Datentypen



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

4.6 FUNKTIONEN HÖHERER ORDNUNG ÜBERBLICK UND MOTIVATION

```
> function logEach(array) {  
  for (var i=0; i<array.length; i++)  
    console.log(array[i]);  
};  
< undefined
```

NORMALE FUNKTION

```
> function forEach(array, action) {  
  for (var i=0; i<array.length; i++)  
    action(array[i]);  
};  
< undefined  
> forEach(["Hunde", "Katzen", "Pferde"], console.log);  
Hunde  
Katzen  
Pferde  
< undefined  
> var sum=0;  
< undefined  
> forEach([1,2,3,4,5], function(number) {  
  sum+=number;  
});  
< undefined  
> sum;  
< 15
```

FUNKTION HÖHERER ORDNUNG

- Funktionen höherer Ordnung (Higher-order Functions) sind Funktionen, die Funktionen als Argumente erhalten oder Funktionen als Ergebnis liefern
- Wichtiges Konzept der funktionalen Programmierung
- Anwendungen
 - Anwenden einer Funktion zur Abbildung aller Elemente eines Feldes auf eine Zielmenge
 - Filtern von Elementen eines Feldes
 - Berechnungen auf den Elementen eines Feldes
- Anwendungen im Bereich Webtechnologien
 - Auswerten von Dokumenten, die von einem Webserver empfangen oder aus der Datenbank gelesen wurden, und ihre Darstellung im Document Object Tree
 - Registrierung von Callback-Funktionen, die beim Eintreten bestimmter Ereignisse ausgeführt werden



http://eloquentjavascript.net/05_higher_order.html

4.6 FUNKTIONEN HÖHERER ORDNUNG

BEISPIELE

FUNKTION, DIE NEUE FUNKTIONEN GENERIERT

```
> function greaterThan(n) {
    return function(m) {return m>n;};
};

< undefined
> var greaterThan10=greaterThan(10);
< undefined
> greaterThan10(11);
< true
```

FUNKTION, DIE ANDERE FUNKTIONEN VERÄNDERT

```
> function noisy(f) {
    return function(arg) {
        console.log("Aufruf mit", arg);
        var val=f(arg);
        console.log("Aufgerufen mit", arg, ", Ergebnis war", val);
        return val;
    };
};

< undefined
> noisy(greaterThan10)(11);
Aufruf mit 11
Aufgerufen mit 11 , Ergebnis war true
< true
```

FUNKTION ZUR EINFÜHRUNG NEUER KONTROLLFLÜSSE

```
> function unless(test, then) {
    if (!test) then();
};

< undefined
> function repeat(times, body) {
    for (var i=0; i<times; i++) body(i);
};

< undefined
> repeat(3, function(n) {
    unless(n%2, function() {
        console.log(n, "is even");
    });
});
0 "is even"
2 "is even"
< undefined
```

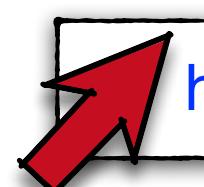
4.6 FUNKTIONEN HÖHERER ORDNUNG

JSON

```
> {
  "Herausgeber": "Xema",
  "Nummer": "1234-5678-9012-3456",
  "Deckung": 2e+6,
  "Waehrung": "EURO",
  "Inhaber": {
    "Name": "Mustermann",
    "Vorname": "Max",
    "maennlich": true,
    "Hobbys": [ "Reiten", "Golfen", "Lesen" ],
    "Alter": 42,
    "Kinder": [],
    "Partner": null
  }
}
< {Herausgeber: "Xema", Nummer: "1234-5678-9012-3456", Deckung: 2000000,
  Waehrung: "EURO", Inhaber: {...}}
```

```
> var string=JSON.stringify({name: "x", born: 1980});
< undefined
> string
< '{"name":"x","born":1980}'
> JSON.parse(string).born;
< 1980
```

- Kompaktes Datenformat in lesbarer Textform zum Datenaustausch zwischen Anwendungen
- JSON (JavaScript Object Notation) ist ähnlich zur JavaScript-Schreibweise für Felder und Objekte
- Ausnahmen und Abweichungen
 - Alle Eigenschaftnamen müssen in Anführungszeichen geschrieben werden
 - Nur einfache Ausdrücke sind erlaubt, d.h. keine Variablen, keine Funktionsaufrufe, keine Berechnungen
 - Keine Kommentare
- JavaScript enthält viele Funktionen, um mit JSON-Dokumenten arbeiten zu können
 - `JSON.stringify()` wandelt JavaScript-Objekte und Felder in JSON um
 - `JSON.parse()` wandelt JSON in JavaScript



http://eloquentjavascript.net/05_higher_order.html

4.6 FUNKTIONEN HÖHERER ORDNUNG

JSON-BEISPIEL

```
> var ANCESTRY_FILE = "[\n " + [\n    {"name": "Carolus Haverbeke", "sex": "m", "born": 1832, "died": 1905, "father": "Carel Haverbeke", "mother": "Maria van Brussel"},\n    {"name": "Emma de Milliano", "sex": "f", "born": 1876, "died": 1956, "father": "Petrus de Milliano", "mother": "Sophia van Damme"},\n    {"name": "Maria de Rycke", "sex": "f", "born": 1683, "died": 1724, "father": "Frederik de Rycke", "mother": "Laurentia van Vlaenderen"},\n    {"name": "Jan van Brussel", "sex": "m", "born": 1714, "died": 1748, "father": "Jacobus van Brussel", "mother": "Joanna van Rooten"},\n    {"name": "Philibert Haverbeke", "sex": "m", "born": 1907, "died": 1997, "father": "Emile Haverbeke", "mother": "Emma de Milliano"},\n    {"name": "Jan Frans van Brussel", "sex": "m", "born": 1761, "died": 1833, "father": "Jacobus Bernardus van Brussel", "mother": "null"},\n    {"name": "Pauwels van Haverbeke", "sex": "m", "born": 1535, "died": 1582, "father": "N. van Haverbeke", "mother": "null"},\n    {"name": "Clara Aernoudts", "sex": "f", "born": 1918, "died": 2012, "father": "Henry Aernoudts", "mother": "Sidonie Coene"},\n    {"name": "Emile Haverbeke", "sex": "m", "born": 1877, "died": 1968, "father": "Carolus Haverbeke", "mother": "Maria Sturm"},\n    {"name": "Lieven de Causmaecker", "sex": "m", "born": 1696, "died": 1724, "father": "Carel de Causmaecker", "mother": "Joanna Claes"},\n    {"name": "Pieter Haverbeke", "sex": "m", "born": 1602, "died": 1642, "father": "Lieven van Haverbeke", "mother": "null"},\n    {"name": "Livina Haverbeke", "sex": "f", "born": 1692, "died": 1743, "father": "Daniel Haverbeke", "mother": "Joanna de Pape"},\n    {"name": "Pieter Bernard Haverbeke", "sex": "m", "born": 1695, "died": 1762, "father": "Willem Haverbeke", "mother": "Petronella Wauters"},\n    {"name": "Lieven van Haverbeke", "sex": "m", "born": 1570, "died": 1636, "father": "Pauwels van Haverbeke", "mother": "Lievijne Jans"},\n    {"name": "Joanna de Causmaecker", "sex": "f", "born": 1762, "died": 1807, "father": "Bernardus de Causmaecker", "mother": "null"},\n    {"name": "Willem Haverbeke", "sex": "m", "born": 1668, "died": 1731, "father": "Lieven Haverbeke", "mother": "Elisabeth Hercke"},\n    {"name": "Pieter Antone Haverbeke", "sex": "m", "born": 1753, "died": 1798, "father": "Jan Francies Haverbeke", "mother": "Petronella de Decker"},\n    {"name": "Maria van Brussel", "sex": "f", "born": 1801, "died": 1834, "father": "Jan Frans van Brussel", "mother": "Joanna de Causmaecker"},\n    {"name": "Angela Haverbeke", "sex": "f", "born": 1728, "died": 1734, "father": "Pieter Bernard Haverbeke", "mother": "Livina de Vrieze"},\n    {"name": "Elisabeth Haverbeke", "sex": "f", "born": 1711, "died": 1754, "father": "Jan Haverbeke", "mother": "Maria de Rycke"},\n    {"name": "Lievijne Jans", "sex": "f", "born": 1542, "died": 1582, "father": "null", "mother": "null"},\n    {"name": "Bernardus de Causmaecker", "sex": "m", "born": 1721, "died": 1789, "father": "Lieven de Causmaecker", "mother": "Livina Haverbeke"},\n    {"name": "Jacoba Lammens", "sex": "f", "born": 1699, "died": 1740, "father": "Lieven Lammens", "mother": "Livina de Vrieze"},\n    {"name": "Pieter de Decker", "sex": "m", "born": 1705, "died": 1780, "father": "Joos de Decker", "mother": "Petronella van de Steene"},\n    {"name": "Joanna de Pape", "sex": "f", "born": 1654, "died": 1723, "father": "Vincent de Pape", "mother": "Petronella Wauters"},\n    {"name": "Daniel Haverbeke", "sex": "m", "born": 1652, "died": 1723, "father": "Lieven Haverbeke", "mother": "Elisabeth Hercke"},\n    {"name": "Lieven Haverbeke", "sex": "m", "born": 1631, "died": 1676, "father": "Pieter Haverbeke", "mother": "Anna van Hecke"},\n    {"name": "Martina de Pape", "sex": "f", "born": 1666, "died": 1727, "father": "Vincent de Pape", "mother": "Petronella Wauters"},\n    {"name": "Jan Francies Haverbeke", "sex": "m", "born": 1725, "died": 1779, "father": "Pieter Bernard Haverbeke", "mother": "Livina de Vrieze"},\n    {"name": "Maria Haverbeke", "sex": "m", "born": 1905, "died": 1997, "father": "Emile Haverbeke", "mother": "Emma de Milliano"},\n    {"name": "Petronella de Decker", "sex": "f", "born": 1731, "died": 1781, "father": "Pieter de Decker", "mother": "Livina Haverbeke"},\n    {"name": "Livina Sierens", "sex": "f", "born": 1761, "died": 1826, "father": "Jan Sierens", "mother": "Maria van Waes"},\n    {"name": "Laurentia Haverbeke", "sex": "f", "born": 1710, "died": 1786, "father": "Jan Haverbeke", "mother": "Maria de Rycke"},\n    {"name": "Carel Haverbeke", "sex": "m", "born": 1796, "died": 1837, "father": "Pieter Antone Haverbeke", "mother": "Livina Sierens"},\n    {"name": "Elisabeth Hercke", "sex": "f", "born": 1632, "died": 1674, "father": "Willem Hercke", "mother": "Margriet de Brabander"},\n    {"name": "Jan Haverbeke", "sex": "m", "born": 1671, "died": 1731, "father": "Lieven Haverbeke", "mother": "Elisabeth Hercke"},\n    {"name": "Anna van Hecke", "sex": "f", "born": 1607, "died": 1670, "father": "Paschiasius van Hecke", "mother": "Martijntken Beelaert"},\n    {"name": "Maria Sturm", "sex": "f", "born": 1835, "died": 1917, "father": "Charles Sturm", "mother": "Seraphina Spelier"},\n    {"name": "Jacobus Bernardus van Brussel", "sex": "m", "born": 1736, "died": 1809, "father": "Jan van Brussel", "mother": "Elisabeth Haverbeke"}\n].join(",\n") + "\n"];\n< undefined\n> var ancestry=JSON.parse(ANCESTRY_FILE);\n< undefined\n> ancestry.length;\n< 39
```

4.6 FUNKTIONEN HÖHERER ORDNUNG FILTERN VON JSON-DATEIEN

```
> function filter(array, test) {
  var passed=[];
  for (var i=0; i<array.length; i++)
    if (test(array[i]))
      passed.push(array[i]);
  return passed;
};

< undefined
> filter(ancestry, function(person) {
  return person.born>1900 && person.born<1925;
});
< ▶ (3) [{}], [{}], [{}]
  ▶ 0: {name: "Philibert Haverbeke", sex: "m", born: 1907, died: 1997, ...
  ▶ 1: {name: "Clara Aernoudts", sex: "f", born: 1918, died: 2012, fath...
  ▶ 2: {name: "Maria Haverbeke", sex: "m", born: 1905, died: 1997, fath...
  length: 3
  __proto__: Array(0)
```

```
> ancestry.filter(function(person) {
  return person.father=="Carel Haverbeke");
};
< ▶ [{}]
  ▶ 0: {name: "Carolus Haverbeke", sex: "m", born: 1832, died: 1905, fa...
  length: 1
  __proto__: Array(0)
```

- Die Funktion `filter(...)` entnimmt einem Feld diejenigen Elemente, welche einen "Test" erfüllen
- `test` ist ein Funktionsargument, das die Referenz auf eine Funktion enthält, welche die Bedingung des Tests enthält
- Hier: Beispielimplementierung von `filter`

- Hier: Verwendung der auf Feldern definierten Standardmethode `filter()`



http://eloquentjavascript.net/05_higher_order.html

4.6 FUNKTIONEN HÖHERER ORDNUNG REDUZIEREN VON JSON-DATEIEN

```
> function reduce(array, combine, start) {  
  var current=start;  
  for (var i=0; i<array.length; i++)  
    current=combine(current, array[i]);  
  return current;  
};  
< undefined  
> reduce([1,2,3,4], function(a,b) {  
  return a+b;  
, 0);  
< 10
```

```
> ancestry.reduce(function(min, cur) {  
  if (cur.born<min.born) return cur;  
  else return min;  
});  
< ► {name: "Pauwels van Haverbeke", sex: "m", born: 1535, died: 1582, fath  
er: "N. van Haverbeke", ...}
```

- Die Funktion `reduce(...)` berechnet aus Feldern einen einzigen Wert, z.B. die Summe aller Elemente oder das Element mit dem kleinsten/größten Wert
 - `combine` ist ein Funktionsargument, welches die zugrundeliegende Berechnung durchführt
 - Hier: Beispielimplementierung von `reduce`
-
- Hier: Verwendung der auf Feldern definierten Standardmethode `reduce()`



http://eloquentjavascript.net/05_higher_order.html

4.6 FUNKTIONEN HÖHERER ORDNUNG

CLOSURES

```
> function wrapValue(n) {  
    var localVariable = n;  
    return function() { return localVariable; };  
};  
var wrap1 = wrapValue(1);  
var wrap2 = wrapValue(2);  
console.log(wrap1());  
console.log(wrap2());  
1  
2  
< undefined
```

- Was passiert wenn Funktionen auf Variablen der übergeordneten Umgebung zugreifen, obwohl diese Umgebung nicht mehr länger existiert?
- Bei Aufruf der anonymen Funktion, die als Rückgabewert von `wrapValue` den Variablen `wrap1` und `wrap2` zugewiesen wird, existiert die übergeordnete Umgebung `wrapValue` nicht mehr
- Trotzdem bleiben die verschiedenen Instanzen von `localVariable` erhalten und liefern beim Aufruf den richtigen Wert
- Ein Closure ist eine Funktion, die den Zugriff auf die Umgebung, die sie erstellt hat, behält, und zwar auch dann, wenn diese Umgebung nicht länger existiert



4.6 FUNKTIONEN HÖHERER ORDNUNG

LAMBDA-AUSDRÜCKE ODER PFEILFUNKTIONEN

```
> var add1=function(a,b) {return a+b};  
< undefined  
> var add2=(a,b)=>a+b;  
< undefined  
> var add3=(a,b)=>{return a+b};  
< undefined  
> var ident=a=>a;  
< undefined  
> var noop=()=>{};  
< undefined
```

```
> [1,2,3,4].filter(function(value) {return value%2==0});  
< ▶ (2) [2, 4]  
> [1,2,3,4].filter(value=>value%2==0);  
< ▶ (2) [2, 4]
```

- Lambda-Ausdrücke oder Pfeilfunktionen sind eine alternative Syntax zu anonymen Funktionsausdrücken
- Verfügbar ab JavaScript-ES6
- Argumente der Funktion werden in Klammern definiert (bei einem Argument sind die Klammern optional), gefolgt von einem Pfeil =>, der die Argumente von einem Ausdruck, einer Anweisung oder einem Anweisungsblock trennt
- Lambda-Ausdrücke werden nicht als Methoden von Objekten verwendet und haben kein eigenes this
- Lambda-Ausdrücke werden hauptsächlich als Argumente an Funktionen höherer Ordnung über- oder zurückgegeben



<https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/Pfeilfunktionen>

4.7 ASYNCHRONE PROGRAMMIERUNG

SYNCHRONE PROGRAMME

```
> function x() {
    console.log("Ergebnis von x()");
};
function y() {
    console.log("Ergebnis von y()");
};
function z() {
    console.log("Ergebnis von z()");
};
x();
console.log("x() ist fertig");
y();
console.log("y() ist fertig");
z();
console.log("z() ist fertig");
Ergebnis von x()
x() ist fertig
Ergebnis von y()
y() ist fertig
Ergebnis von z()
z() ist fertig
< undefined
```

- Synchrone Programme werden durch die Laufzeitumgebung Zeile für Zeile abgearbeitet
- Laufzeitumgebung wartet bis die gegenwärtige Zeile abgearbeitet ist und springt dann zur nächsten Zeile
- Bei Funktionsaufruf wird der Befehlszeiger auf den Speicherbereich mit der ersten Anweisung innerhalb der Funktion gesetzt und dort die synchrone Ausführung fortgesetzt
- Bei Rückkehr aus einer Funktion wird ab der nächsten Zeile nach dem ursprünglichen Funktionsaufruf fortgesetzt
- Mit rein synchroner Programmierung kann nicht auf Ereignisse reagiert werden
- Synchrone Programme führen oft zu Blockierungen, z.B. auf Interaktionen mit einer Webseite wird nicht reagiert, weil die Laufzeitumgebung "mit anderen Dingen beschäftigt" ist

4.7 ASYNCHRONE PROGRAMMIERUNG

ASYNCHRONE PROGRAMME

Callback \rightarrow asynchronität

```
> function x(callback) {
    setTimeout(function() {
        console.log("Ergebnis von x()");
        callback();
    }, 2000);
};

function y(callback) {
    setTimeout(function() {
        console.log("Ergebnis von y()");
        callback();
    }, 1000);
};

function z(callback) {
    setTimeout(function() {
        console.log("Ergebnis von z()");
        callback();
    }, 3000);
};

x(()=>{
    console.log("x() ist fertig");
    y(()=>{
        console.log("y() ist fertig");
        z(()=> {
            console.log("z() ist fertig");
        });
    });
});

< undefined
Ergebnis von x()
x() ist fertig
Ergebnis von y()
y() ist fertig
Ergebnis von z()
z() ist fertig
```

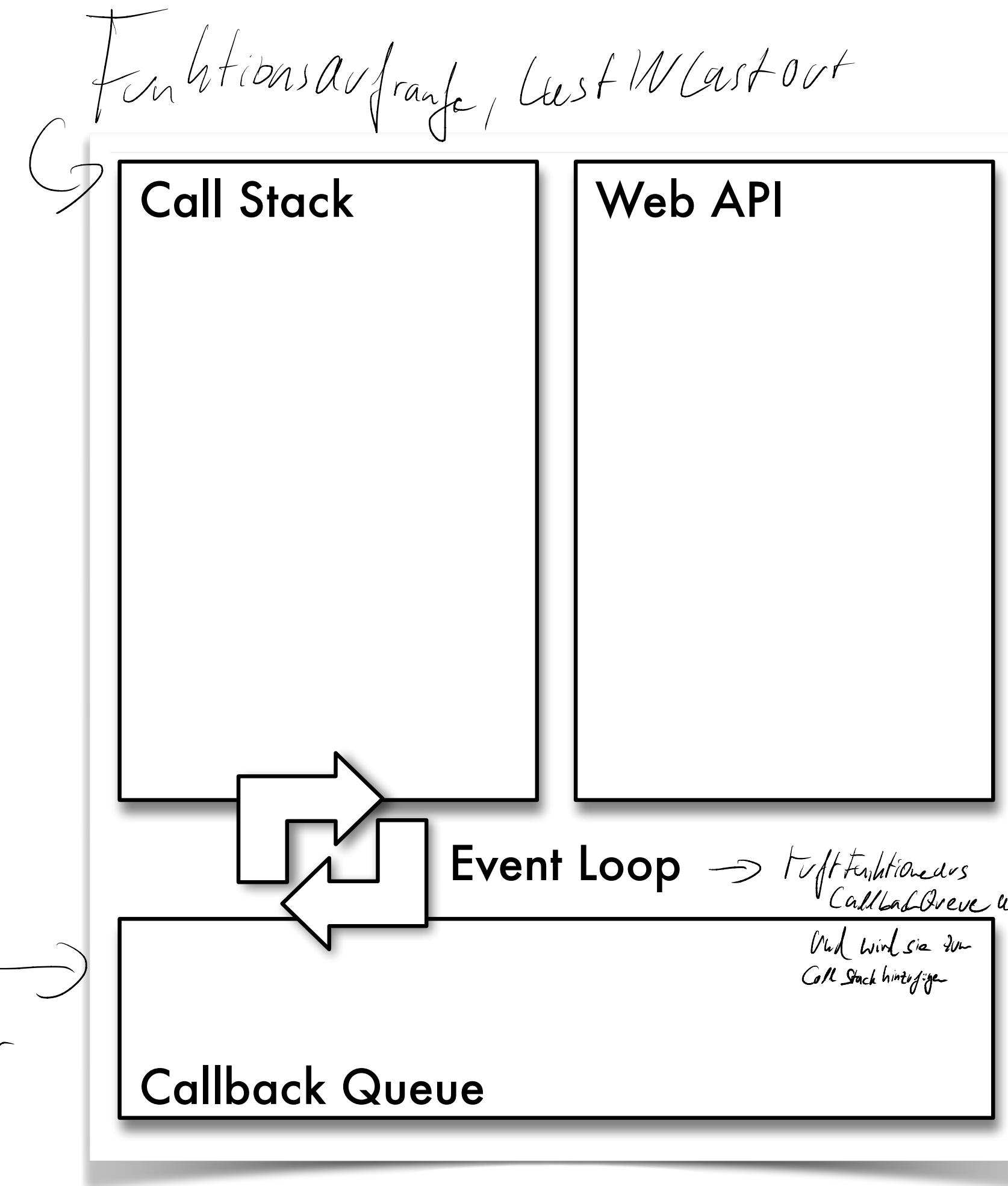
- Bei asynchronen Programmen werden Teile des Programms aus der synchronen Ausführung herausgenommen und deren Ausführung auf einen bestimmten oder unbestimmten späteren Zeitpunkt verschoben
- Spätere Ausführung der asynchronen Programmteile wird durch Eintreten vordefinierter Ereignisse ausgelöst oder findet statt, wenn die Laufzeitumgebung "nichts anderes zu tun hat"
- Asynchrone Programmierung wird eingesetzt, wenn ein zeitlich vorhersehbarer Ablauf des Programms nicht gegeben ist
- Asynchrone Programmierung führt zu reaktiven Programmen die i.d.R. weniger blockieren
- Callbacks sind Funktionen, die der Laufzeitumgebung zur asynchronen Ausführung übergeben werden

ASYNCHRONE MECHANISMEN IM WEBBROWSER

- Web APIs (XMLHttpRequest, Fetch API,...)
- setTimeout, setInterval
- Promises

4.7 ASYNCHRONE PROGRAMMIERUNG

ARCHITEKTUR VON JS RUNTIMES



- Eine JavaScript Laufzeitumgebung (Runtime Environment) interpretiert JavaScript-Programme und führt sie aus
- Wichtige Elemente der Laufzeitumgebung sind Call Stack, Callback Queue und Event Loop
- Laufzeitumgebung im Browser interagiert mit Web APIs zur asynchronen Ausführung von Programmen
- Call Stack ist ein Stapspeicher der den Ausführungskontext von Funktionen nach dem Last-In-Last-Out-Prinzip (last-in-first-out) verwaltet
last In first Out
- Callback Queue ist eine Warteschlange, welche den zur asynchronen Ausführung bestimmten Code verwaltet
- Event Loop überprüft kontinuierlich die Einträge im Call Stack - sind keine vorhanden, übergibt sie dem Call Stack Einträge aus der Callback Queue zur Ausführung
- JavaScript Runtime ist single-threaded, d.h. sie verfügt nur über einen Call Stack und die Ausführung mehrerer Threads ist nicht vorgesehen (Ausnahme: Web Worker API)

4.7 ASYNCHRONE PROGRAMMIERUNG

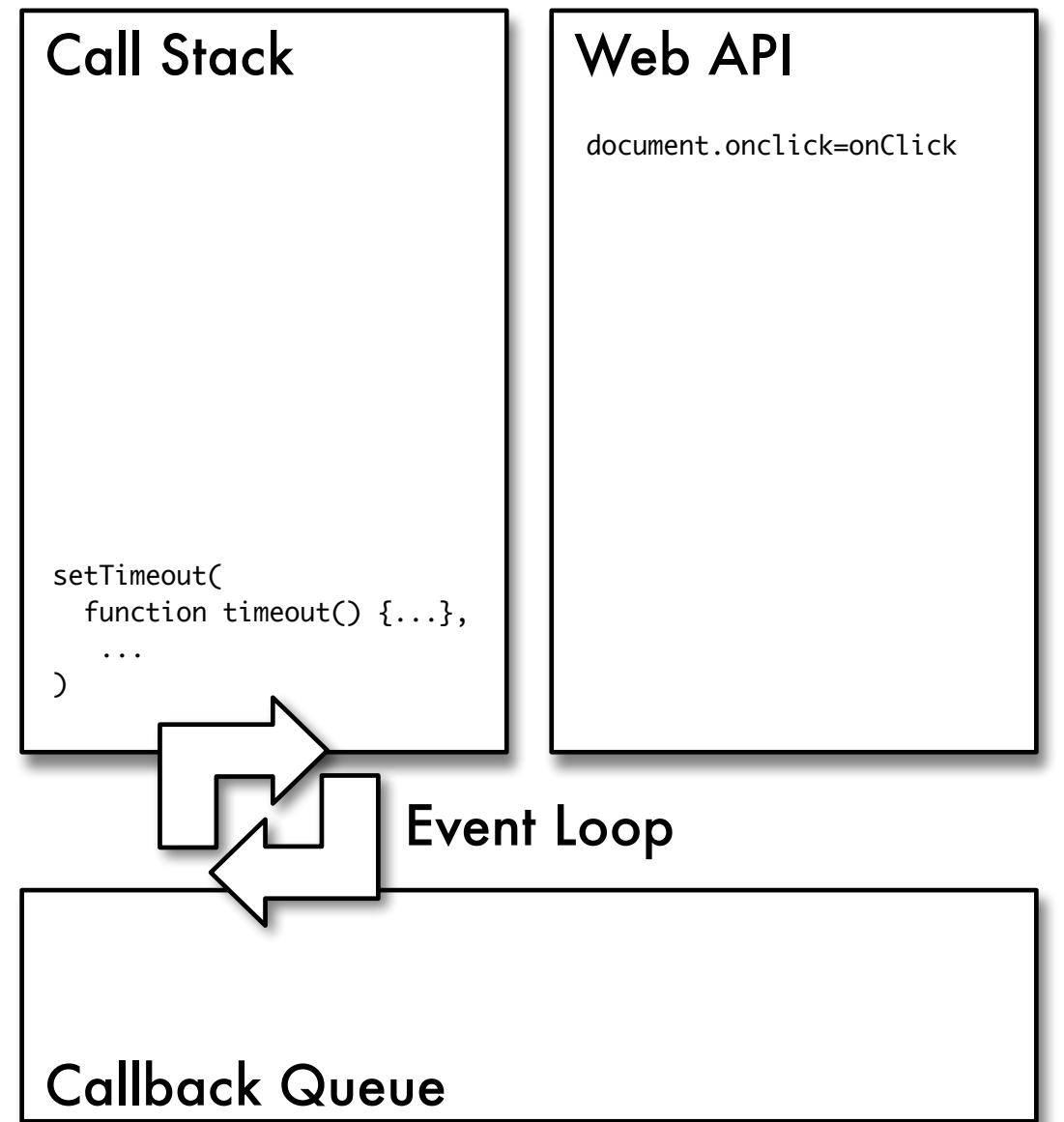
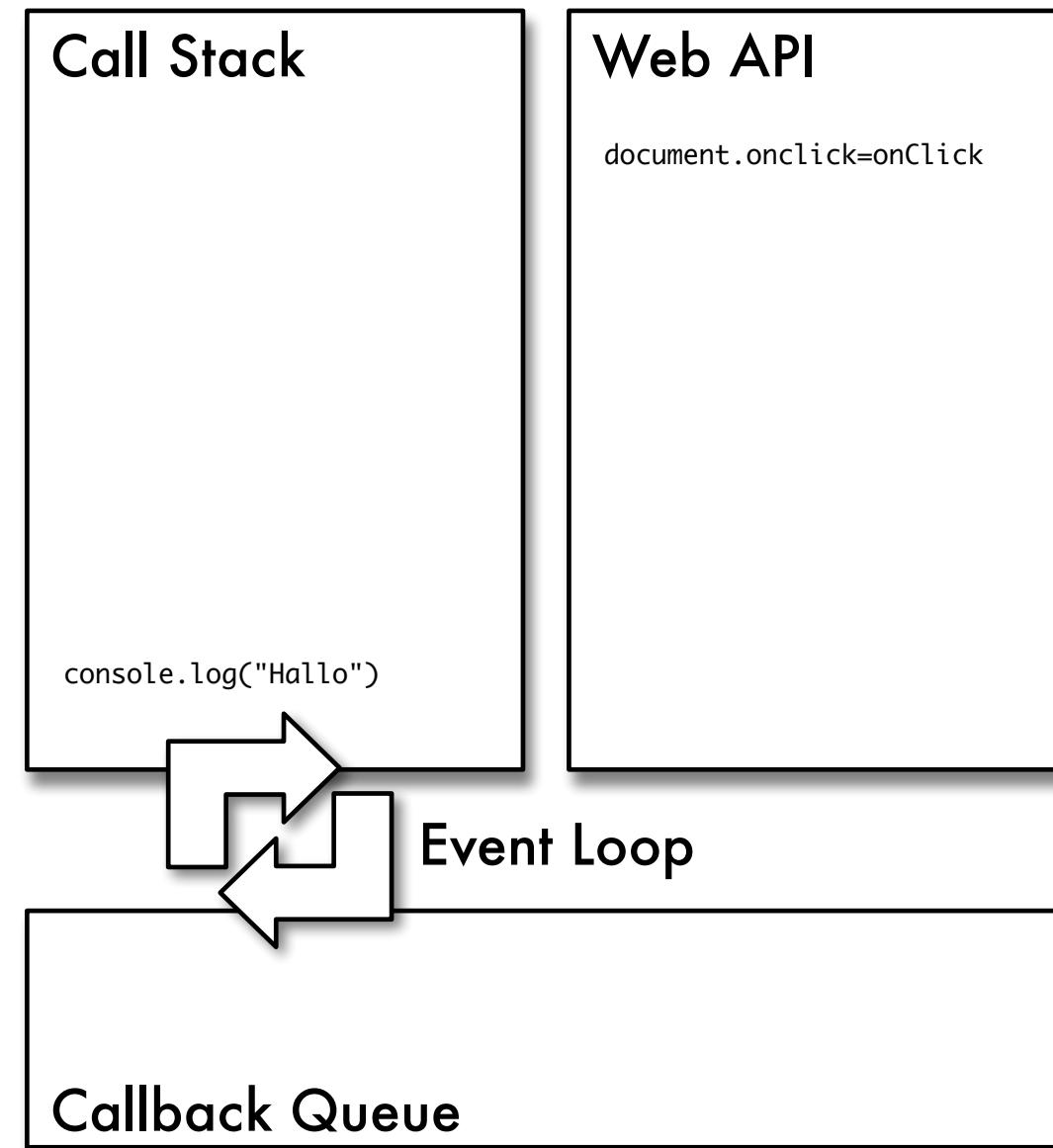
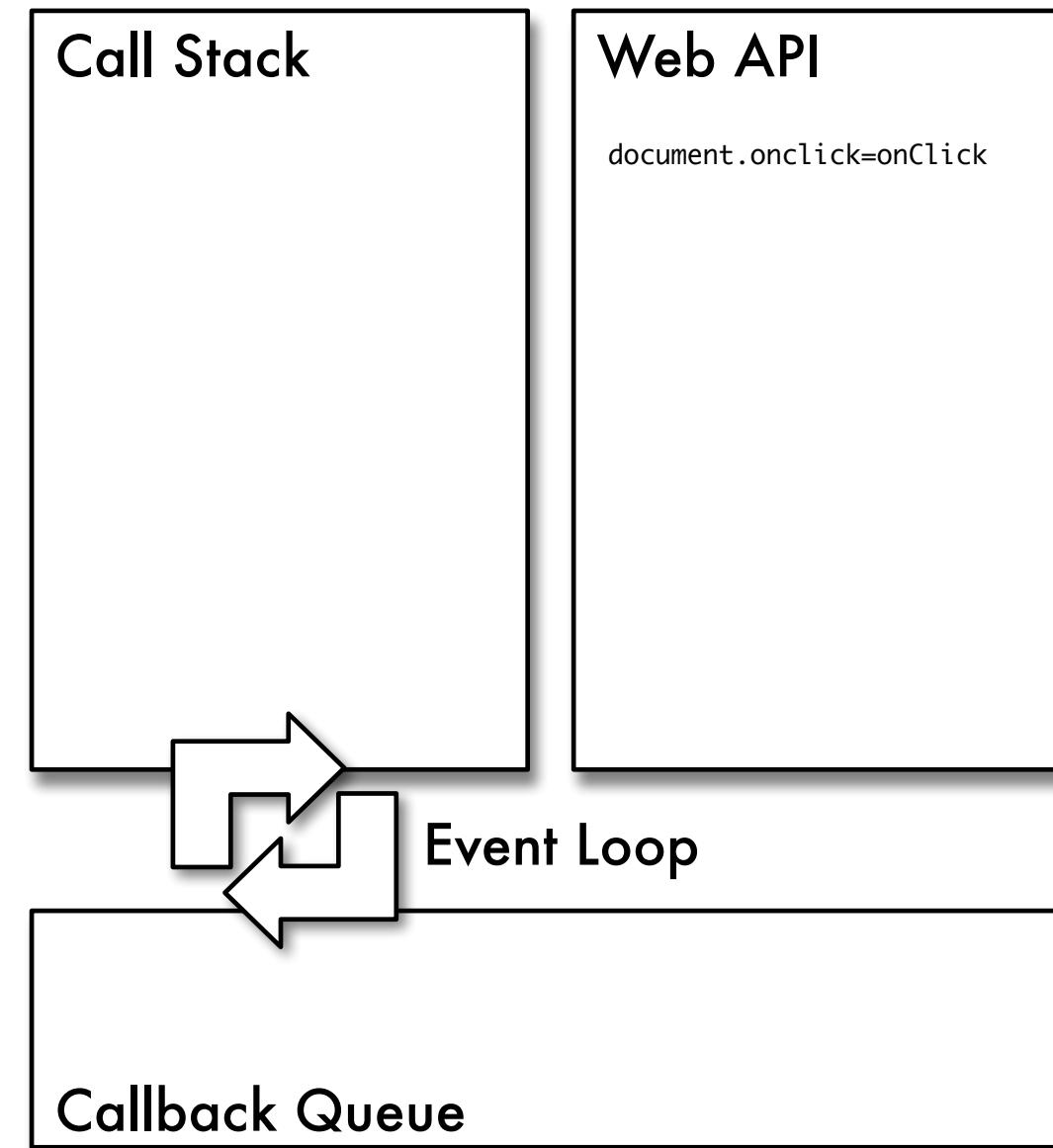
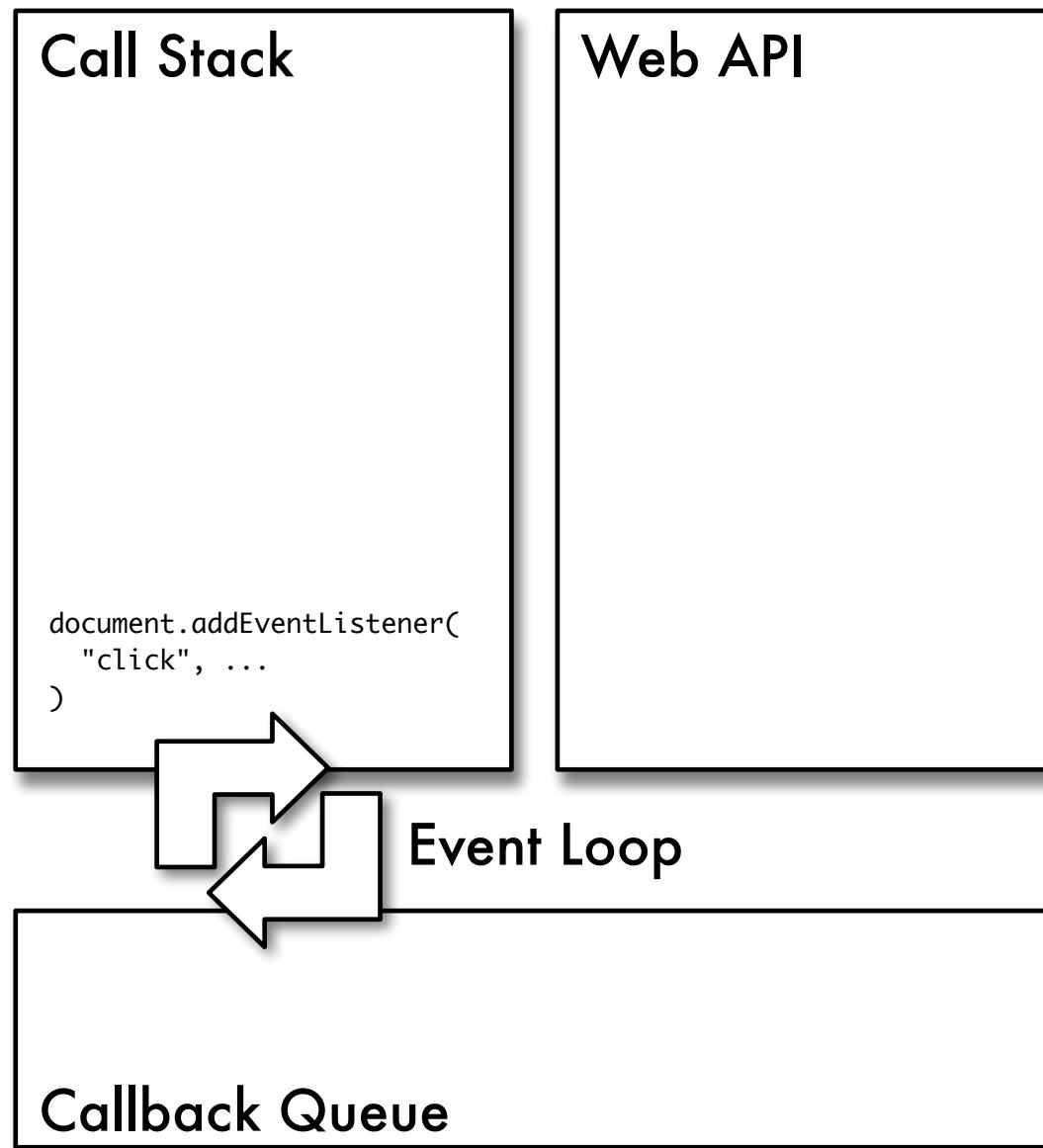
ARBEITSWEISE DER EVENT LOOP (I)

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  }  
);  
  
console.log("Hallo");  
  
setTimeout(function timeout() {  
  console.log("Drücke die Maustaste!");  
}, 10000);  
  
console.log("Synchrone Ausführung beendet");
```

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  }  
);  
  
console.log("Hallo");  
  
setTimeout(function timeout() {  
  console.log("Drücke die Maustaste!");  
}, 10000);  
  
console.log("Synchrone Ausführung beendet");
```

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  }  
);  
  
console.log("Hallo");  
  
setTimeout(function timeout() {  
  console.log("Drücke die Maustaste!");  
}, 10000);  
  
console.log("Synchrone Ausführung beendet");
```

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  }  
);  
  
console.log("Hallo");  
  
setTimeout(function timeout() {  
  console.log("Drücke die Maustaste!");  
}, 10000);  
  
console.log("Synchrone Ausführung beendet");  
  
Hallo
```



4.7 ASYNCHRONE PROGRAMMIERUNG

ARBEITSWEISE DER EVENT LOOP (II)

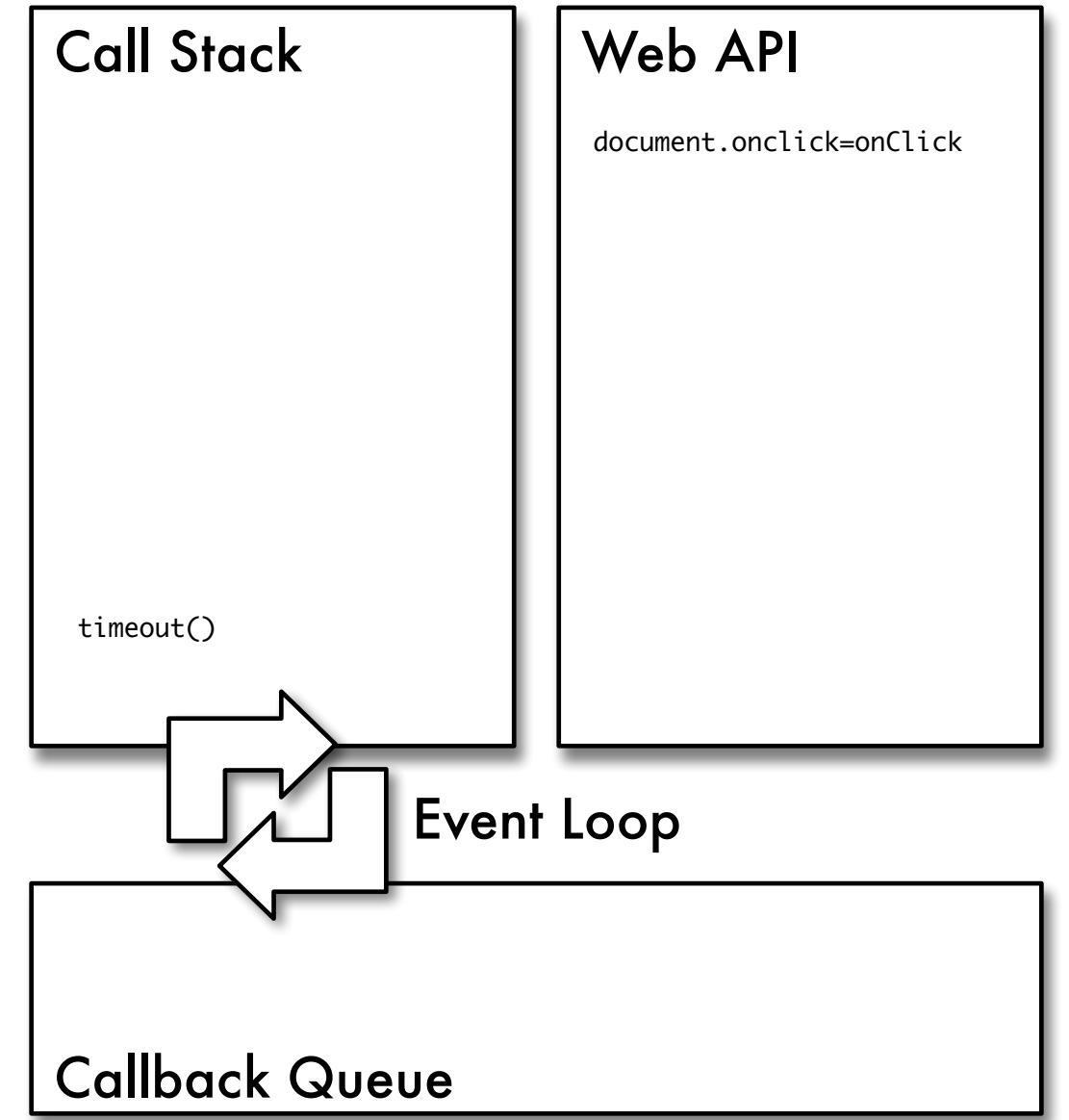
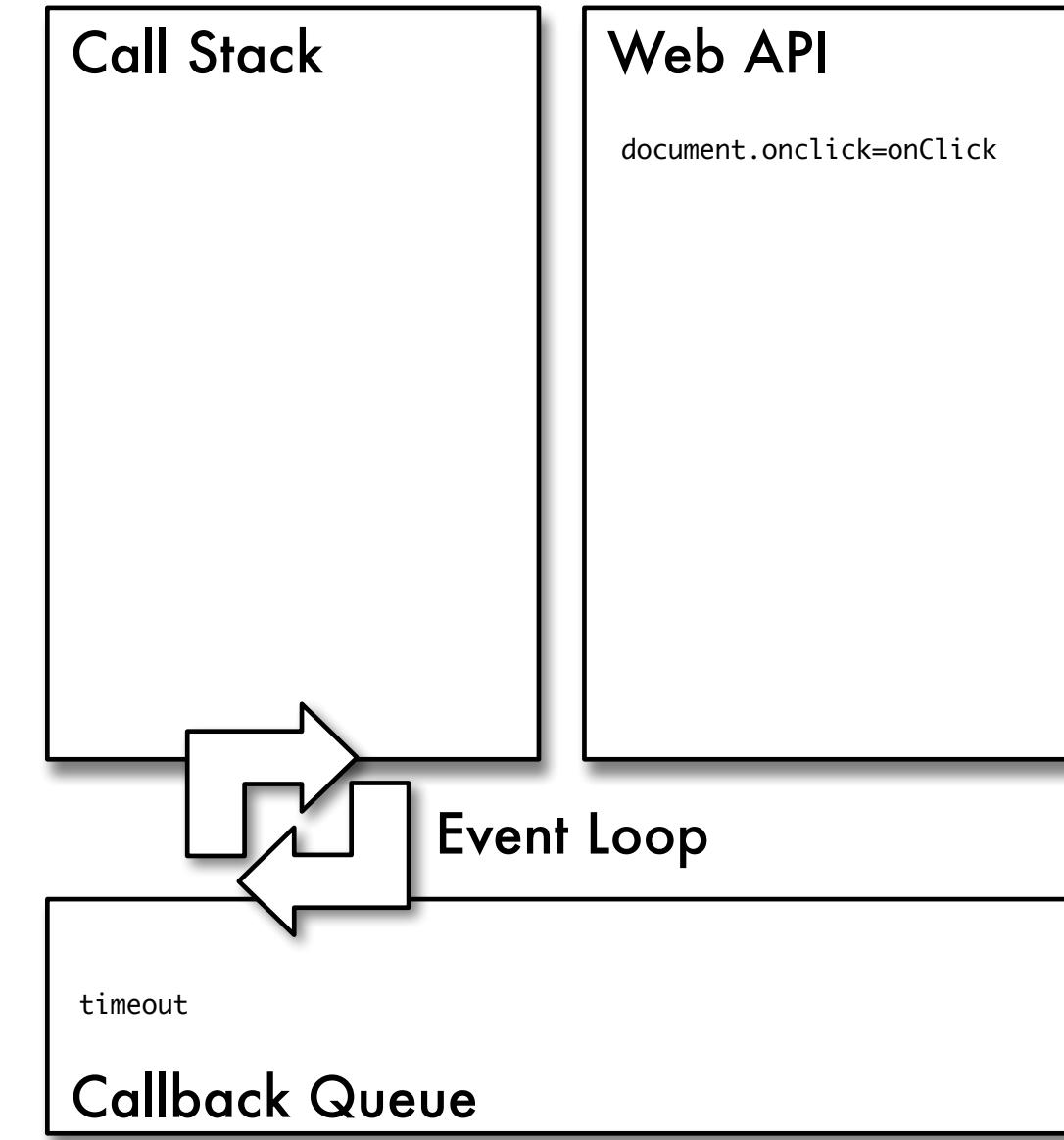
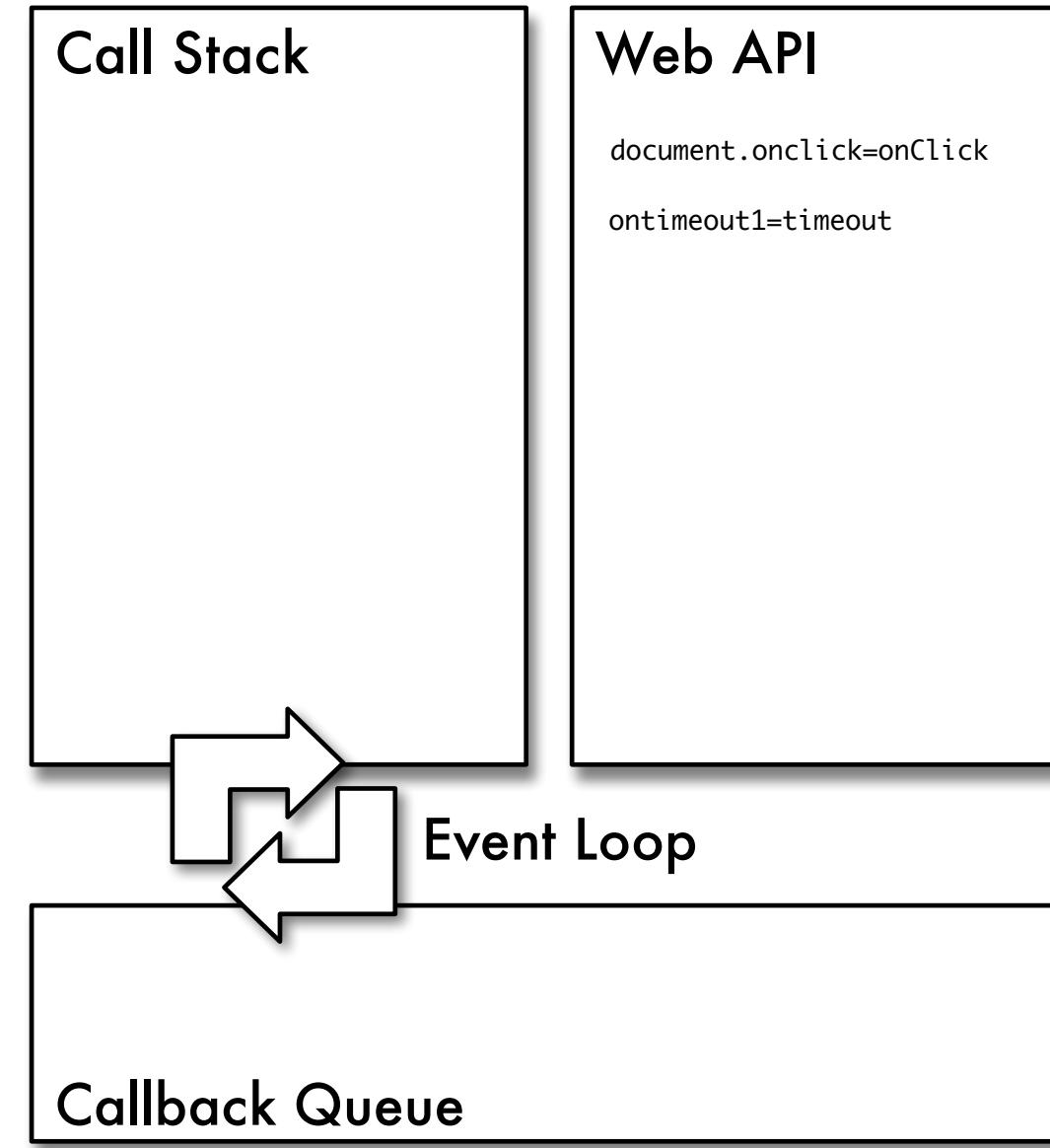
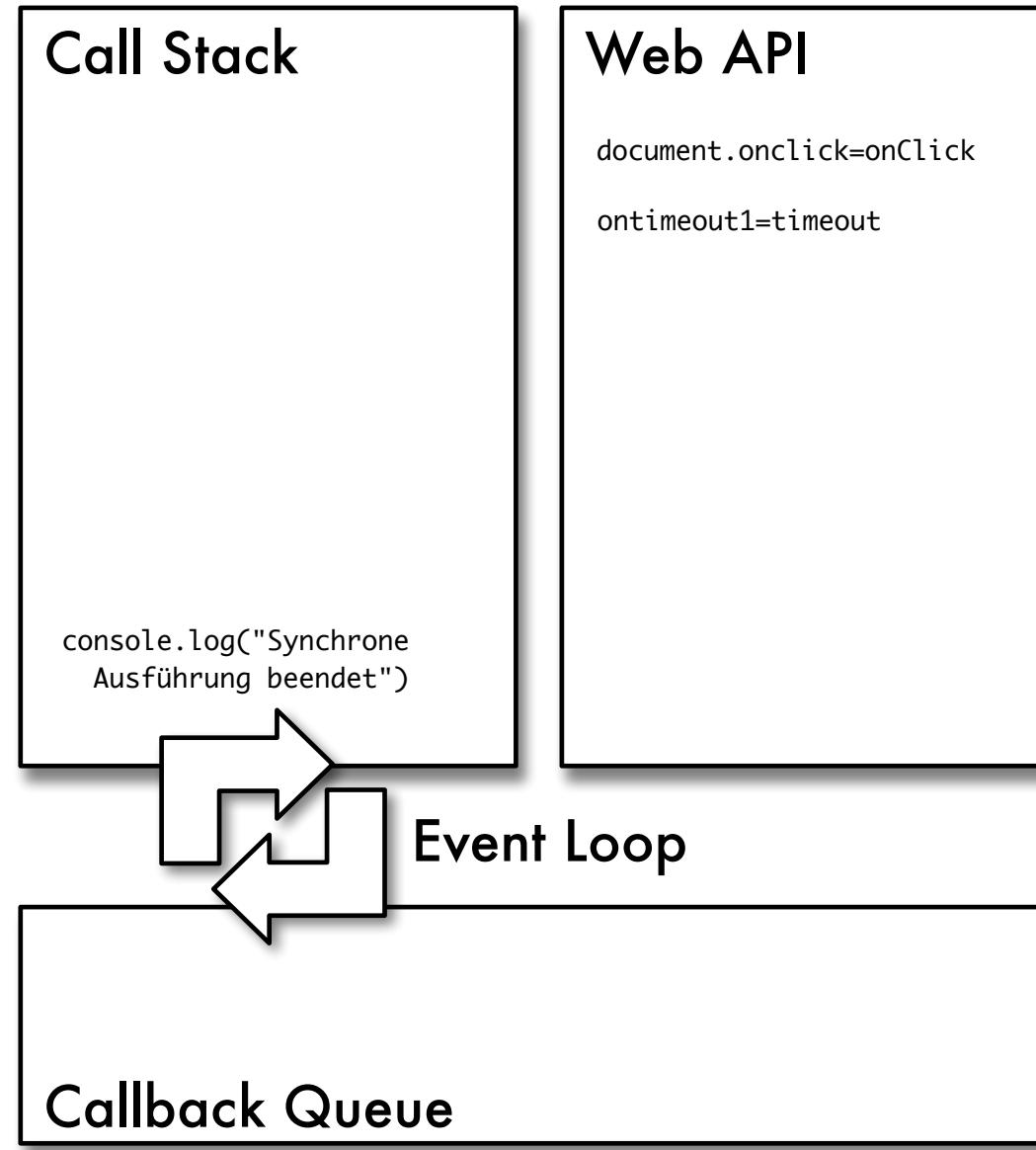
```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
console.log("Hallo");  
  
setTimeout(function timeout() {  
  console.log("Drücke die Maustaste!");  
}, 10000);  
  
console.log("Synchrone Ausführung beendet");  
Hallo
```

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
console.log("Hallo");  
  
setTimeout(function timeout() {  
  console.log("Drücke die Maustaste!");  
}, 10000);  
  
console.log("Synchrone Ausführung beendet");  
Hallo  
Synchrone Ausführung beendet  
<- undefined
```

ca. 10 Sekunden
nach Starten des
Programms

```
Haloo  
Synchrone Ausführung beendet  
<- undefined
```

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
console.log("Hallo");  
  
setTimeout(function timeout() {  
  console.log("Drücke die Maustaste!");  
}, 10000);  
  
console.log("Synchrone Ausführung beendet");  
Hallo  
Synchrone Ausführung beendet  
<- undefined
```



4.7 ASYNCHRONE PROGRAMMIERUNG

ARBEITSWEISE DER EVENT LOOP (III)

```
> document.addEventListener(
  "click", function onClick() {
    setTimeout(function timer() {
      console.log("Du hast mit der Maus geklickt");
    }, 2000);
  }
);

console.log("Hallo");

setTimeout(function timeout() {
  console.log("Drücke die Maustaste!");
}, 10000);

console.log("Synchrone Ausführung beendet");
Hallo
Synchrone Ausführung beendet
< undefined
```

```
> document.addEventListener(
  "click", function onClick() {
    setTimeout(function timer() {
      console.log("Du hast mit der Maus geklickt");
    }, 2000);
  }
);

console.log("Hallo");

setTimeout(function timeout() {
  console.log("Drücke die Maustaste!");
}, 10000);

console.log("Synchrone Ausführung beendet");
Hallo
Synchrone Ausführung beendet
< undefined
Drücke die Maustaste!
```

Nutzer klickt

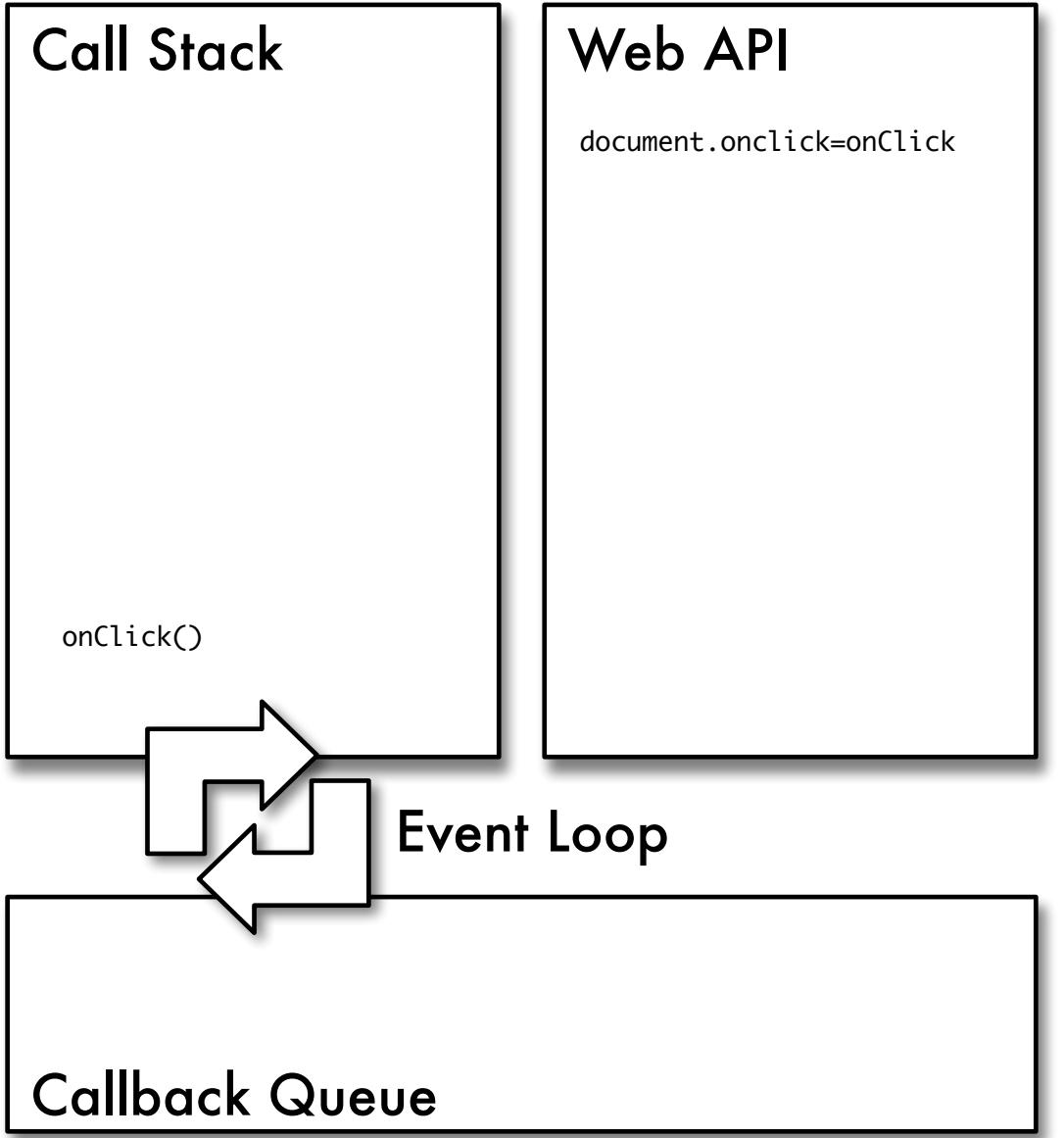
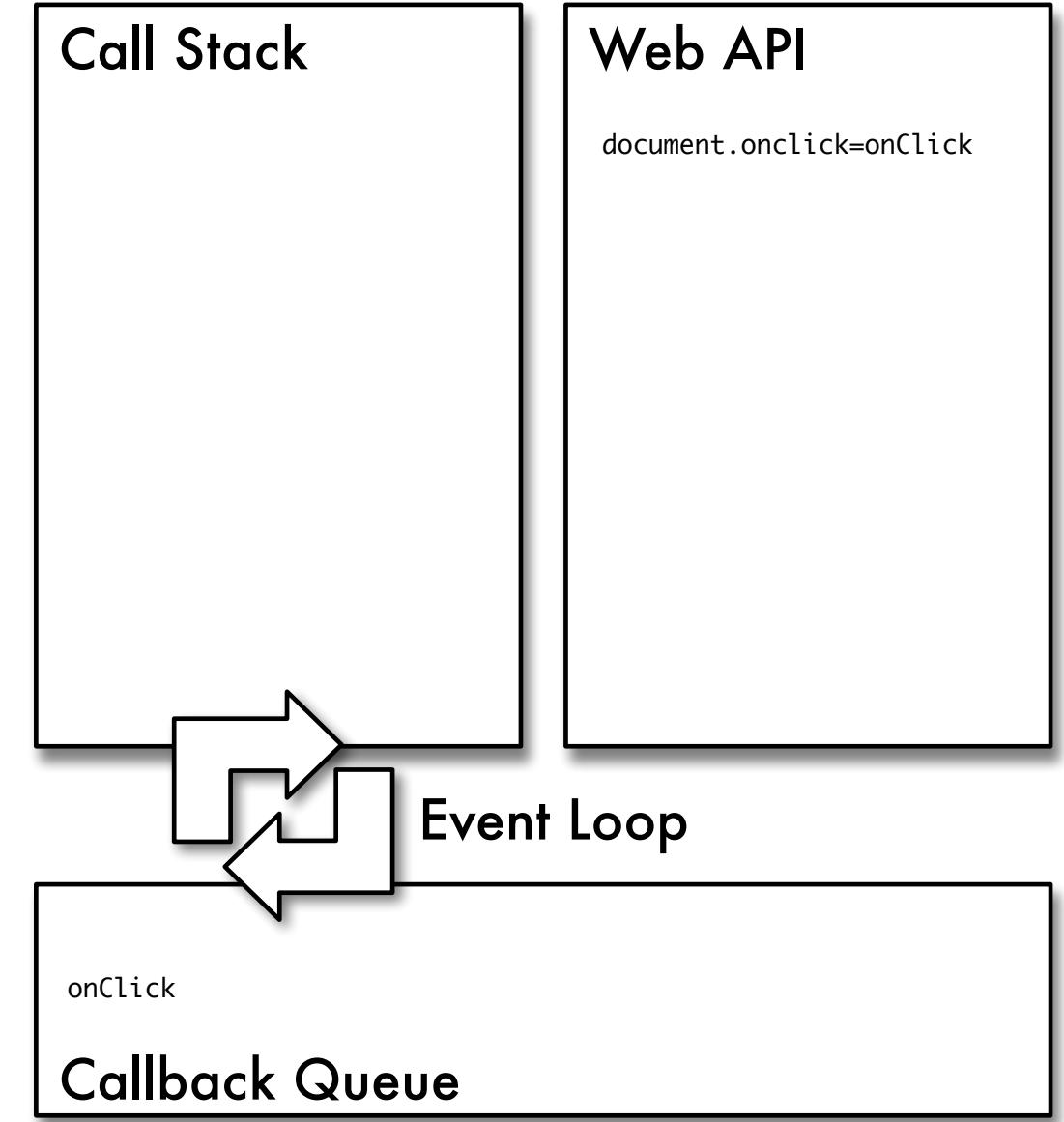
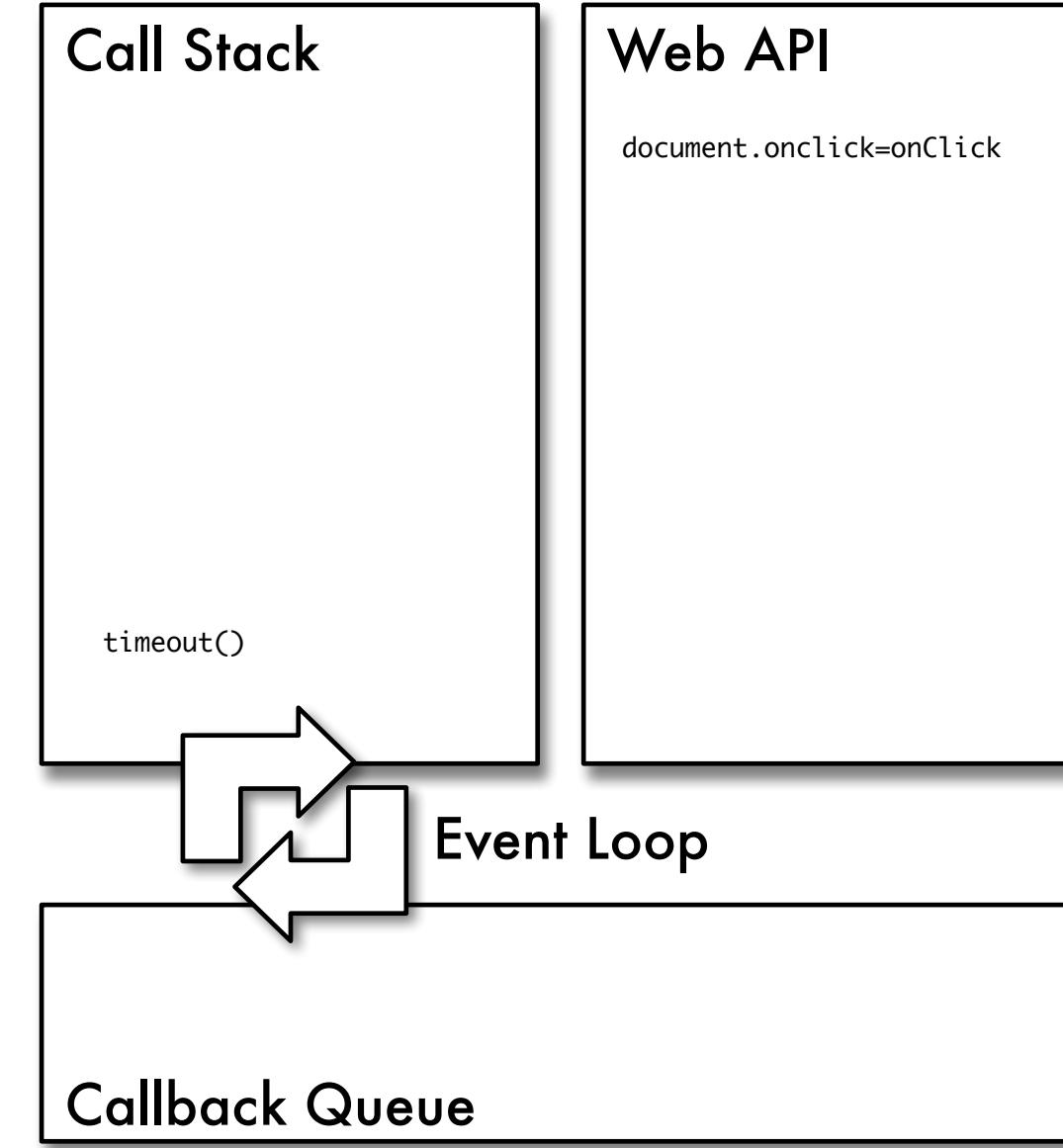
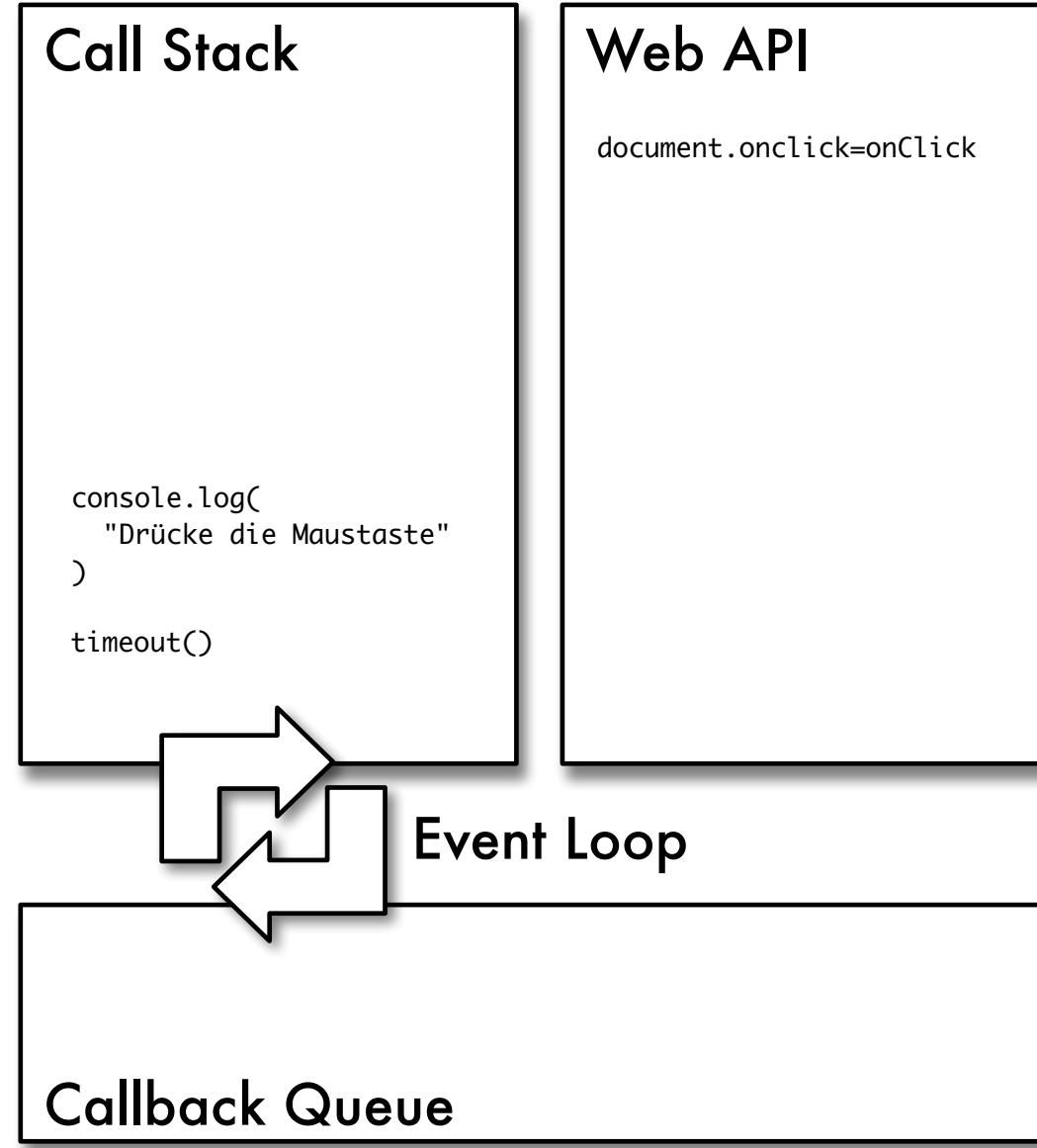
```
Hallo
Synchrone Ausführung beendet
< undefined
Drücke die Maustaste!
```

```
> document.addEventListener(
  "click", function onClick() {
    setTimeout(function timer() {
      console.log("Du hast mit der Maus geklickt");
    }, 2000);
  }
);

console.log("Hallo");

setTimeout(function timeout() {
  console.log("Drücke die Maustaste!");
}, 10000);

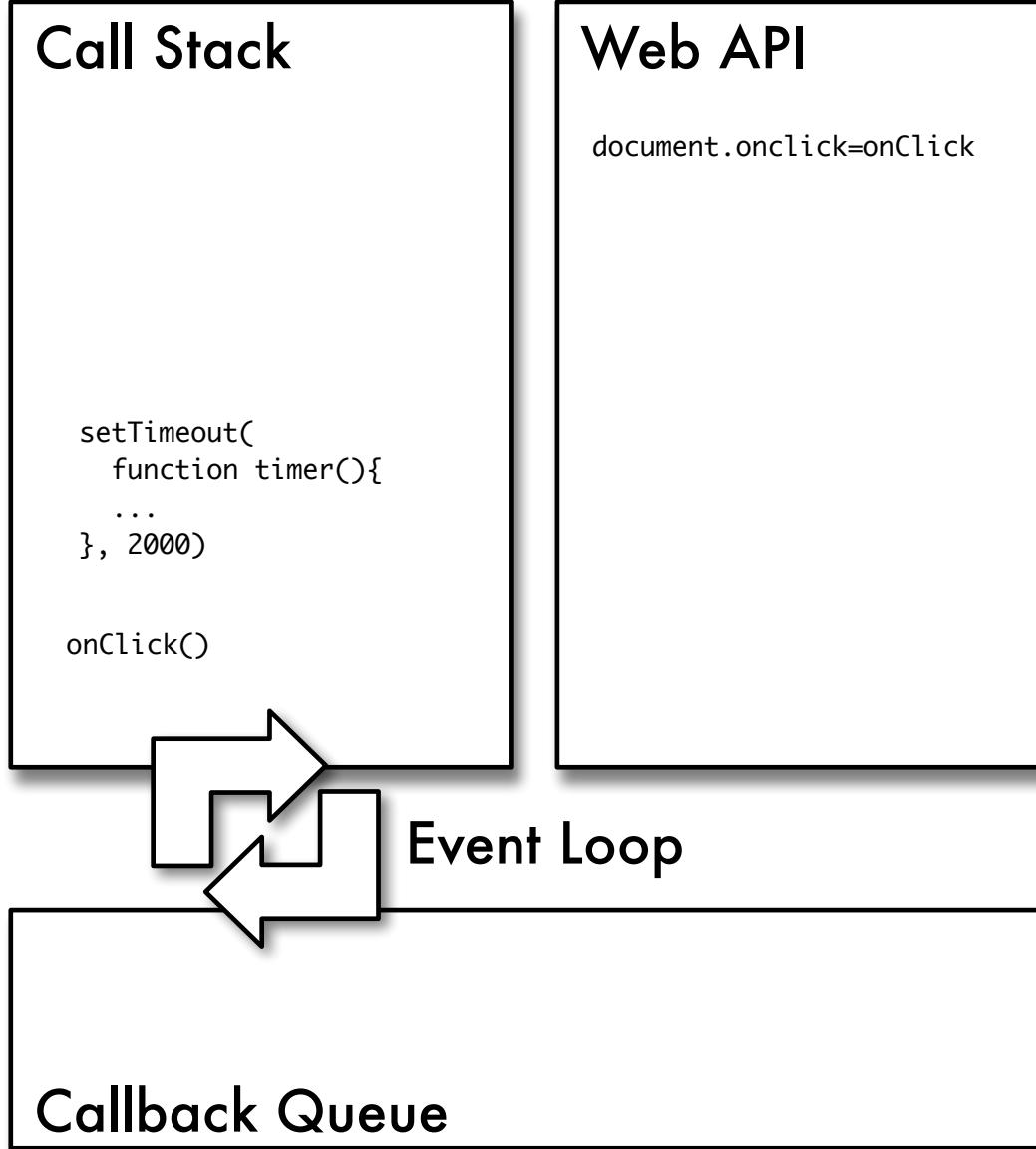
console.log("Synchrone Ausführung beendet");
Hallo
Synchrone Ausführung beendet
< undefined
Drücke die Maustaste!
```



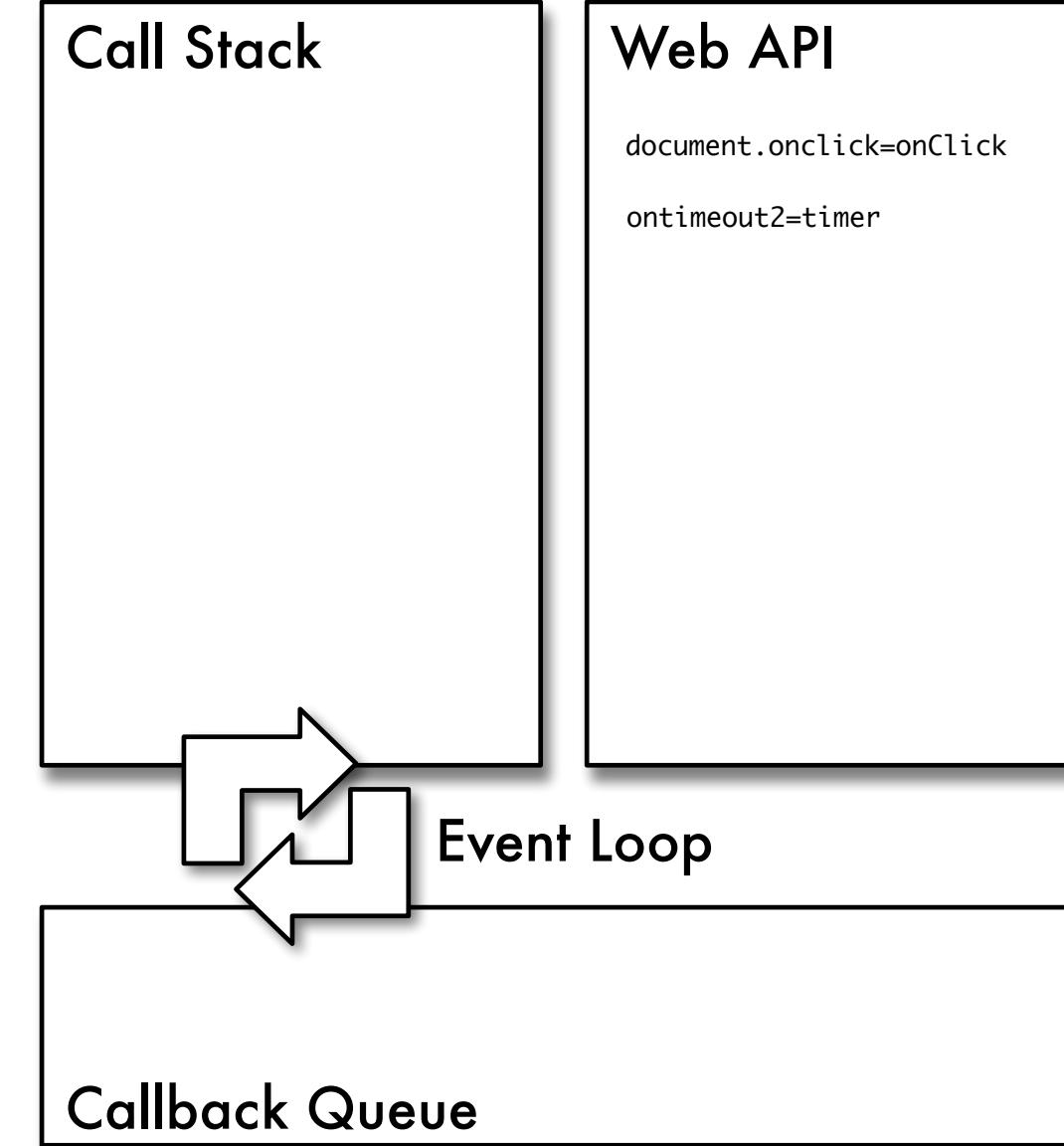
4.7 ASYNCHRONE PROGRAMMIERUNG

ARBEITSWEISE DER EVENT LOOP (IV)

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
  console.log("Hallo");  
  
  setTimeout(function timeout() {  
    console.log("Drücke die Maustaste!");  
  }, 10000);  
  
  console.log("Synchrone Ausführung beendet");  
  Hallo  
  Synchrone Ausführung beendet  
< undefined  
  Drücke die Maustaste!
```

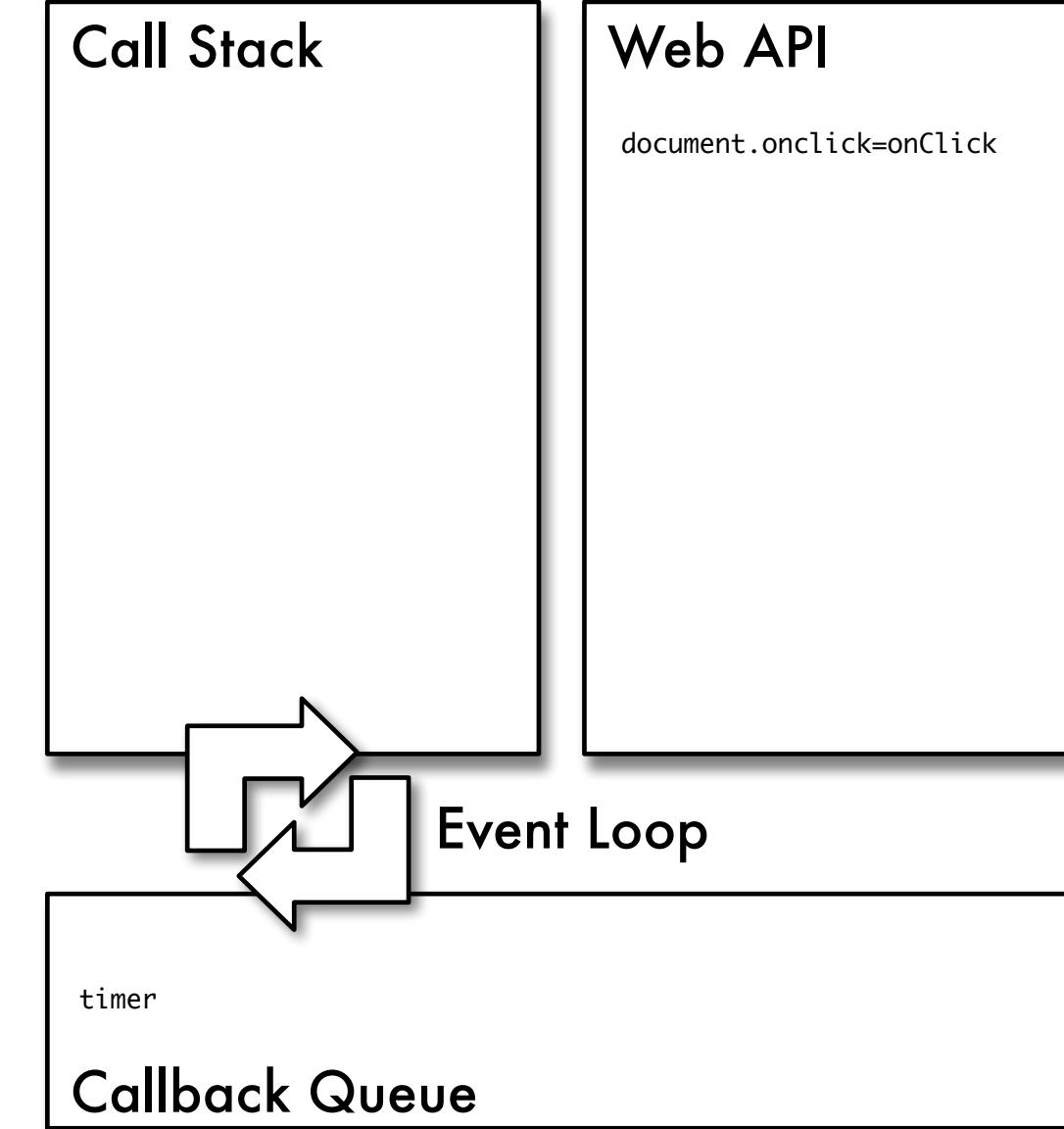


```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
  console.log("Hallo");  
  
  setTimeout(function timeout() {  
    console.log("Drücke die Maustaste!");  
  }, 10000);  
  
  console.log("Synchrone Ausführung beendet");  
  Hallo  
  Synchrone Ausführung beendet  
< undefined  
  Drücke die Maustaste!
```

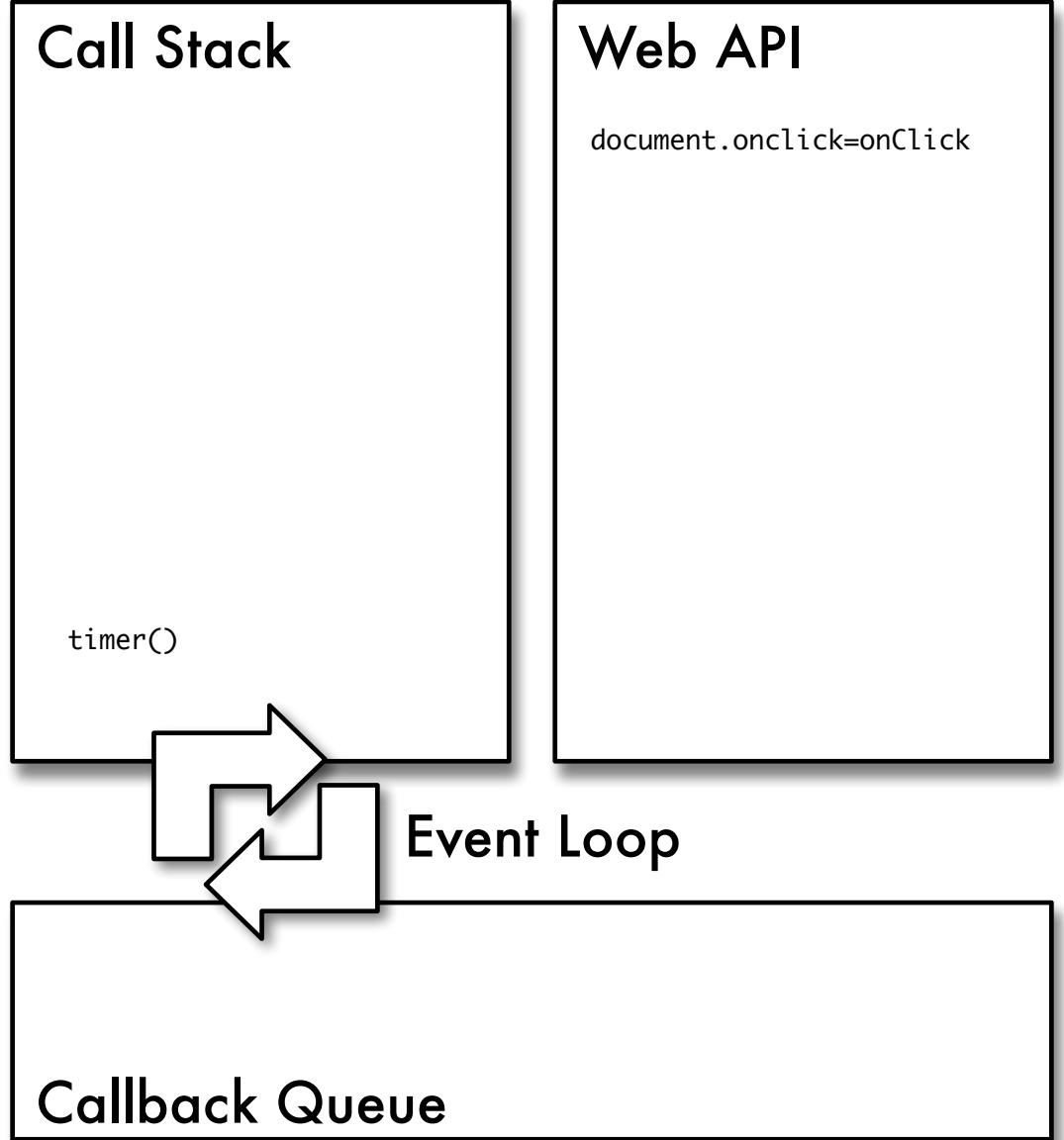


ca. 2 Sekunden nach
Klicken des Nutzers

```
Haloo  
Synchrone Ausführung beendet  
< undefined  
Drücke die Maustaste!
```



```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
  console.log("Hallo");  
  
  setTimeout(function timeout() {  
    console.log("Drücke die Maustaste!");  
  }, 10000);  
  
  console.log("Synchrone Ausführung beendet");  
  Hallo  
  Synchrone Ausführung beendet  
< undefined  
  Drücke die Maustaste!  
  Du hast mit der Maus geklickt
```



4.7 ASYNCHRONE PROGRAMMIERUNG

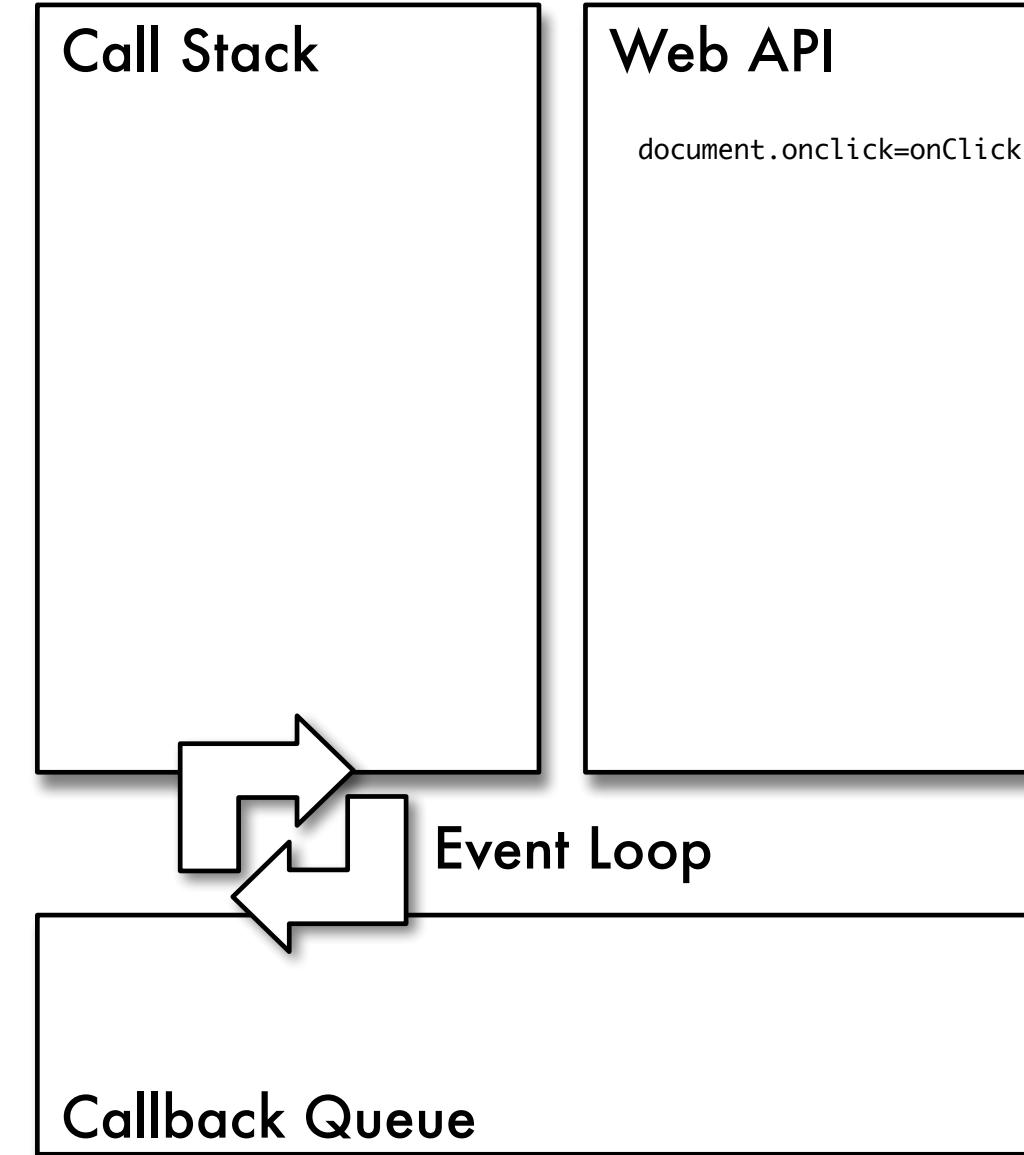
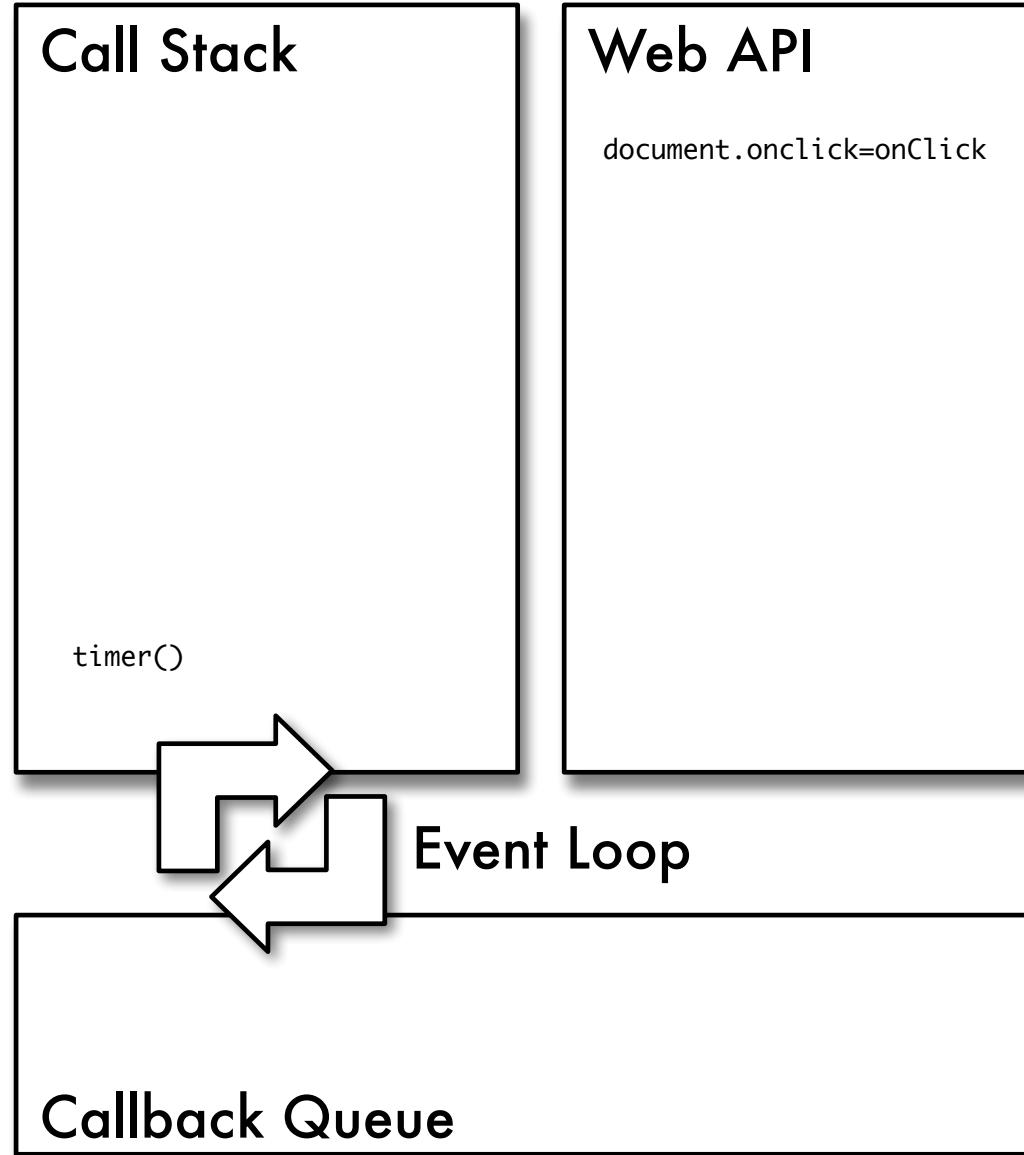
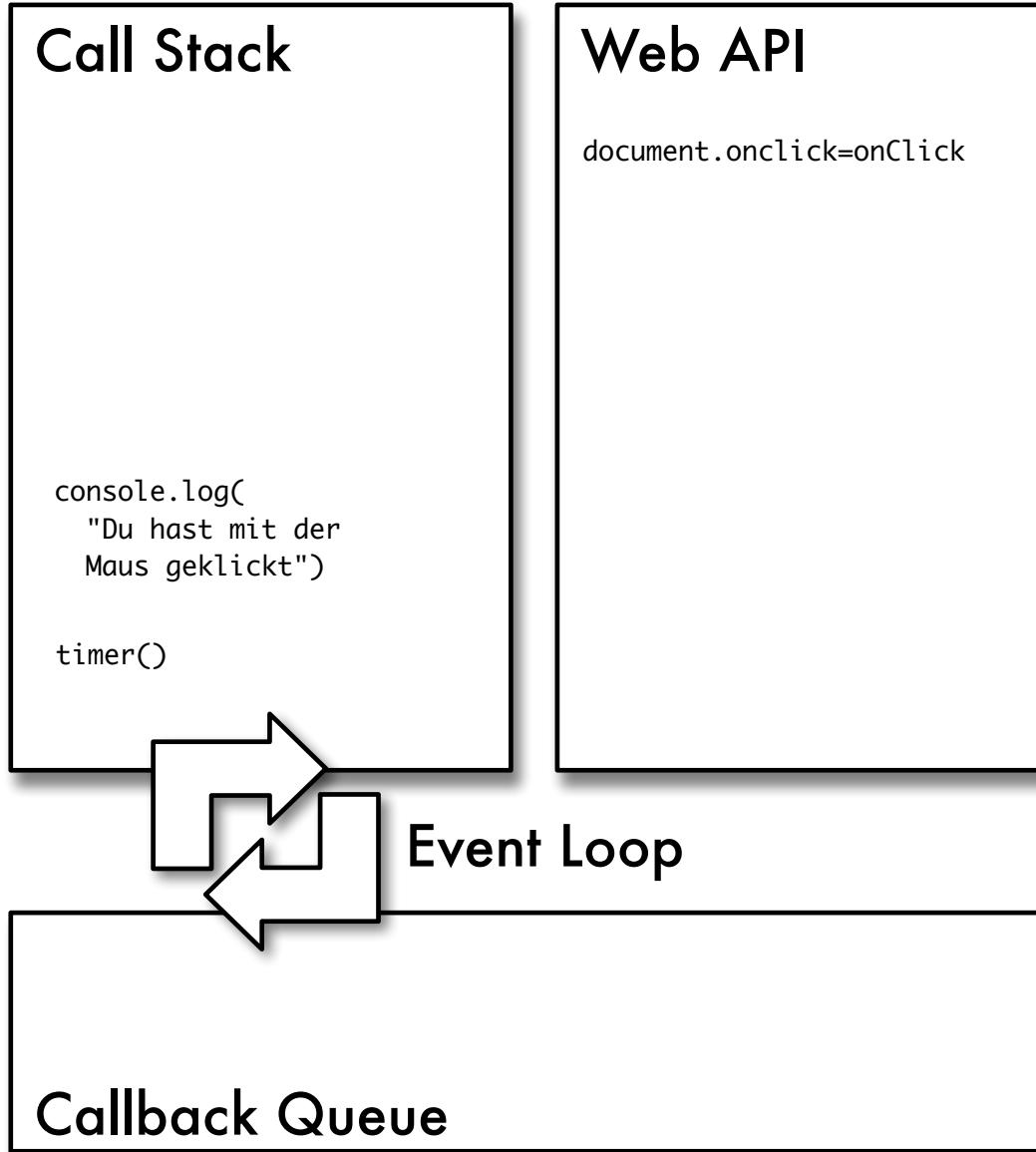
ARBEITSWEISE DER EVENT LOOP (V)

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
  console.log("Hallo");  
  
  setTimeout(function timeout() {  
    console.log("Drücke die Maustaste!");  
  }, 10000);  
  
  console.log("Synchrone Ausführung beendet");  
  Hallo  
  Synchrone Ausführung beendet  
< undefined  
  Drücke die Maustaste!
```

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
  console.log("Hallo");  
  
  setTimeout(function timeout() {  
    console.log("Drücke die Maustaste!");  
  }, 10000);  
  
  console.log("Synchrone Ausführung beendet");  
  Hallo  
  Synchrone Ausführung beendet  
< undefined  
  Drücke die Maustaste!  
  Du hast mit der Maus geklickt
```

```
> document.addEventListener(  
  "click", function onClick() {  
    setTimeout(function timer() {  
      console.log("Du hast mit der Maus geklickt");  
    }, 2000);  
  };  
  
  console.log("Hallo");  
  
  setTimeout(function timeout() {  
    console.log("Drücke die Maustaste!");  
  }, 10000);  
  
  console.log("Synchrone Ausführung beendet");  
  Hallo  
  Synchrone Ausführung beendet  
< undefined  
  Drücke die Maustaste!  
  Du hast mit der Maus geklickt
```

Letztendlich : von API → Callback Queue
→ Funktion zu Callback



4.7 ASYNCHRONE PROGRAMMIERUNG DIE CALLBACK-HÖLLE

```
> function x(callback) {
    setTimeout(function() {
        console.log("Ergebnis von x()");
        callback();
    }, 2000);
};

function y(callback) {
    setTimeout(function() {
        console.log("Ergebnis von y()");
        callback();
    }, 1000);
};

function z(callback) {
    setTimeout(function() {
        console.log("Ergebnis von z()");
        callback();
    }, 3000);
};

x(()=>{
    console.log("x() ist fertig");
    y(()=>{
        console.log("y() ist fertig");
        z(()=> {
            console.log("z() ist fertig");
        });
    });
});

< undefined
Ergebnis von x()
x() ist fertig
Ergebnis von y()
y() ist fertig
Ergebnis von z()
z() ist fertig
```

- Oftmals aktivieren Callbacks andere Callbacks und es muss eine gewisse Reihenfolge bei der Bearbeitung der Callbacks eingehalten werden
- Als Callback-Hölle (Callback Hell) werden mehrere ineinander verschachtelte Callbacks bezeichnet, deren Code nur noch sehr schwer nachvollziehbar ist
- Lösung: Promises

4.7 ASYNCHRONE PROGRAMMIERUNG

LÖSUNG: PROMISES

Konsumfunktionen

Ausführungs-funktion

ERFOLGSFALL (FULFILLED)

Ergbnisobj.

```
> var promise=new Promise(function(resolve, reject) {
    setTimeout(()=>resolve("Fertig! Hier ist das Ergebnis"), 3000);
}).then(
    result=>console.log(result),
    error=>console.log(error)
).finally(()=>console.log("Bearbeitung beendet"));
< undefined
Fertig! Hier ist das Ergebnis
Bearbeitung beendet
```

Ergebnisobjekt

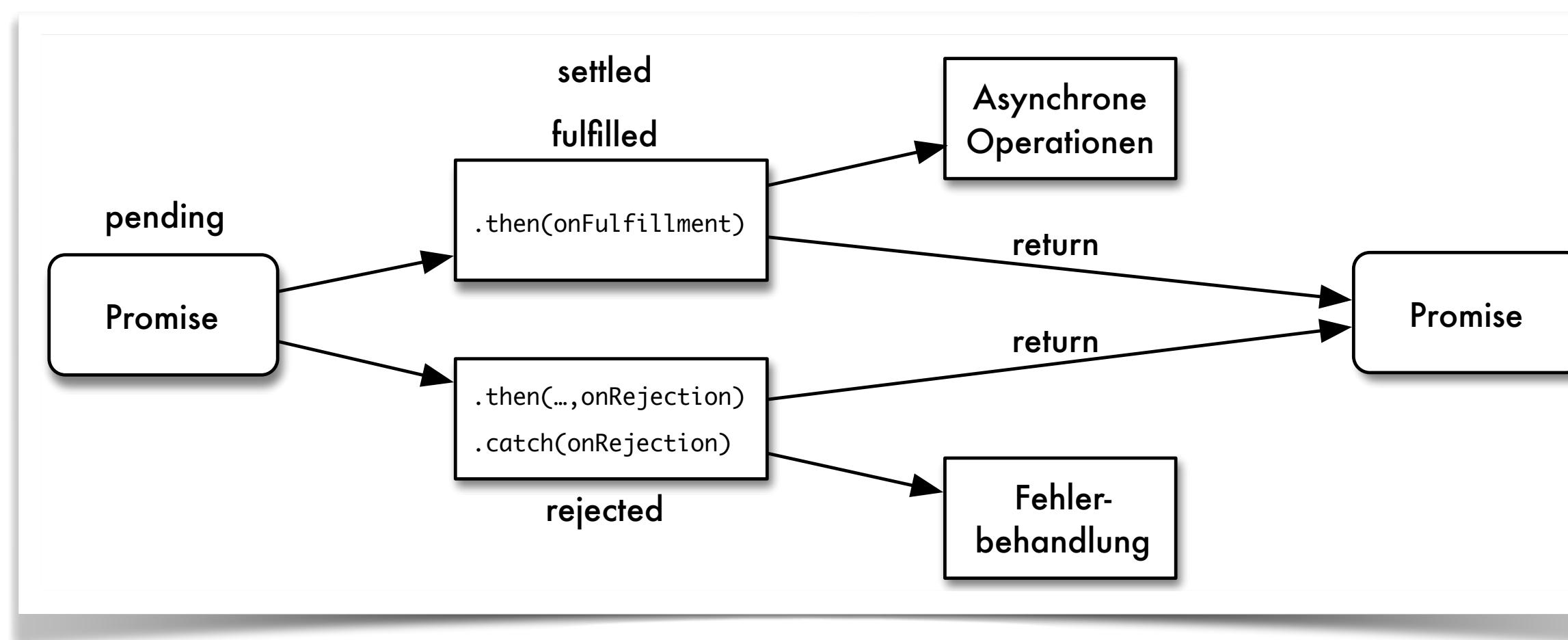
FEHLERFALL (REJECTED)

```
> var promise=new Promise(function(resolve, reject) {
    setTimeout(()=>reject(new Error("Fertig! Hier ist der Fehler")), 3000);
}).then(
    result=>console.log(result),
    error=>console.log(error)
).finally(()=>console.log("Bearbeitung beendet"));
< undefined
Error: Fertig! Hier ist der Fehler
at <anonymous>:2:27
Bearbeitung beendet
```

Fehlerobjekt

- Ein Promise ist ein "Versprechen" welches in Zukunft entweder erfüllt (fulfilled) oder nicht eingehalten (rejected) wird
- Promises sind Objekte, an die beim Aufruf ihres Konstruktors eine Ausführungsfunction (Executor Function) übergeben wird
- Ausführungsfunction wird synchron bei Instanziierung des Promise ausgeführt, definiert aber i.d.R. asynchronen Code in Form von Callbacks, die an Funktionen höherer Ordnung, z.B. Web APIs, übergeben werden
- Ausführungsfunction endet mit Aufruf von `resolve` und Übergabe eines Ergebnisobjekts mit `reject` und Übergabe eines Fehlerobjekts
- Das vom Konstruktor erzeugte Promise-Objekt enthält eine `then`-Methode, welcher als Argument eine Konsumfunktion (Consumer Function) für den Erfolgsfall und eine für den Fehlerfall übergeben wird
- Konsumfunktionen werden in eine Job Queue im Erfolgsfall bzw. Fehlerfall aufgenommen und von der Event Loop mit dem Ergebnis- bzw. Fehlerobjekt aufgerufen

4.7 ASYNCHRONE PROGRAMMIERUNG DER PROMISE LIFE CYCLE



- Zustände des Promise-Objekts
 - pending: initialer Zustand, Ausführungsfunktion noch nicht abgeschlossen
 - fulfilled: Ausführungsfunktion wurde erfolgreich abgeschlossen
 - rejected: Ausführungsfunktion ist gescheitert
 - settled: Ausführungsfunktion mit fulfilled oder rejected beendet
- Konsumfunktion im Fehlerfalle kann alternativ auch der catch-Methode übergeben werden (anstelle als zweites Argument in then), optional
- An finally übergebene Callbacks werden unabhängig von fulfilled und rejected ausgeführt und sind keine Konsumfunktionen, das sie keine Argumente haben

4.7 ASYNCHRONE PROGRAMMIERUNG

CHAINED PROMISES

```
> var myPromise=new Promise((resolve, reject)=> {
  setTimeout(()=>{
    resolve("Immer");
  }, 3000) ↳ wird die Laufzeit übergeben
}).then((wert)=>wert+" und immer") → jedes then generiert ein neues Promise
  .then((wert)=>wert+" und immer wieder")
  .then((wert)=>wert+" wiederhole ich mich.")
  .then((wert)=>console.log(wert))
  .then((wert)=>{throw new Error("Achtung Fehler")})
  .catch((err)=>console.error(err));
< undefined
      Immer und immer und immer wieder wiederhole ich mich.
✖ ► Error: Achtung Fehler
      at <anonymous>:9:24
```

- then, catch und finally geben nach Beendigung ein neues Promise-Objekt zurück, wodurch verkettete Promises (Chained Promises) möglich werden
- Konsumfunktionen in verketteten then-Methoden werden in der vorgegebenen Reihenfolge asynchron abgearbeitet
- Wird in einer der Konsumfunktionen ein Fehler ausgelöst, wird then-Kette abgebrochen und der Fehler optional durch eine an catch übergebene Konsumfunktion behandelt

4.7 ASYNCHRONE PROGRAMMIERUNG

ASYNC

```
> async function meineFunktion() {
    return new Promise((resolve)=>resolve(1));
}
<- undefined
> meineFunktion().then((wert)=>console.log("Ergebnis: "+wert));
    Ergebnis: 1
<- ►Promise {<fulfilled>: undefined}
```

```
> async function meineFunktion() {
    return 1;
}
<- undefined
> meineFunktion().then((wert)=>console.log("Ergebnis: "+wert));
    Ergebnis: 1
<- ►Promise {<fulfilled>: undefined}
```

```
> async function meineFunktion() {
    console.log("nix");
}
<- undefined
> meineFunktion().then((wert)=>console.log("Ergebnis: "+wert));
    nix
    Ergebnis: undefined
<- ►Promise {<fulfilled>: undefined}
```

- Präfix `async` vor Funktionen zeigt an, dass die Funktion ein Promise als Rückgabeargument liefert
- Der Rückgabewert kann explizit als Promise deklariert werden, wird der Rückgabewert mit einem anderen Datentyp deklariert wird er automatisch in ein Promise eingepackt

4.7 ASYNCHRONE PROGRAMMIERUNG

AWAIT

```
> var meinPromise=()=>{
    return new Promise((resolve, reject)=>{
        setTimeout(()=>{
            resolve("Ich wurde aufgelöst.")
        }, 3000);
    });
<- undefined
> async function ohneAwait() {
    var wert=meinPromise();
    console.log("Ergebnis: "+wert);
}
<- undefined
> async function mitAwait() {
    var wert=await meinPromise();
    console.log("Ergebnis: "+wert);
}
<- undefined
> ohneAwait()
Ergebnis: [object Promise]
<- ▶ Promise {<fulfilled>: undefined}
> mitAwait()
<- ▶ Promise {<pending>}
Ergebnis: Ich wurde aufgelöst.
```

Await : Wartet bis die Berechnung des Promises fertig ist
und ersetzt die Referenz des Promises durch das Promis Ergebnis
↳ eine Variable wird direkt ein Ergebnis vermittelt

- Präfix await vor einem Promise zeigt an, dass auf die Finalisierung des Promise gewartet wird
- Warten geschieht asynchron, d.h. die Event Loop erledigt zwischenzeitlich andere Aufgaben
- await ersetzt zudem die Referenz auf das Promise durch das Promise-Ergebnis
- await darf nur in async-Funktionen angewendet werden

FORTSETZUNG FOLGT...



VORLESUNG

Prof. Dr. Axel Küpper

TU Berlin | T-Labs | Fachgebiet *Service-centric Networking*
Ernst-Reuter-Platz 7 | 10587 Berlin | Germany

 axel.kuepper@tu-berlin.de

 <https://twitter.com/kuepp>

 <https://www.linkedin.com/in/axelkuepper/>

 <http://www.snet.tu-berlin.de/kuepper>

ÜBUNGSLEITER

- Thomas Cory
- Sanjeet Raj Pandey
- Christian René Sechting

TUTOREN

- Nastassia Lukyanovich
- Maximilian Oliver Fisch
- Leonhardt Frederik Hollatz
- Adrian Siebing