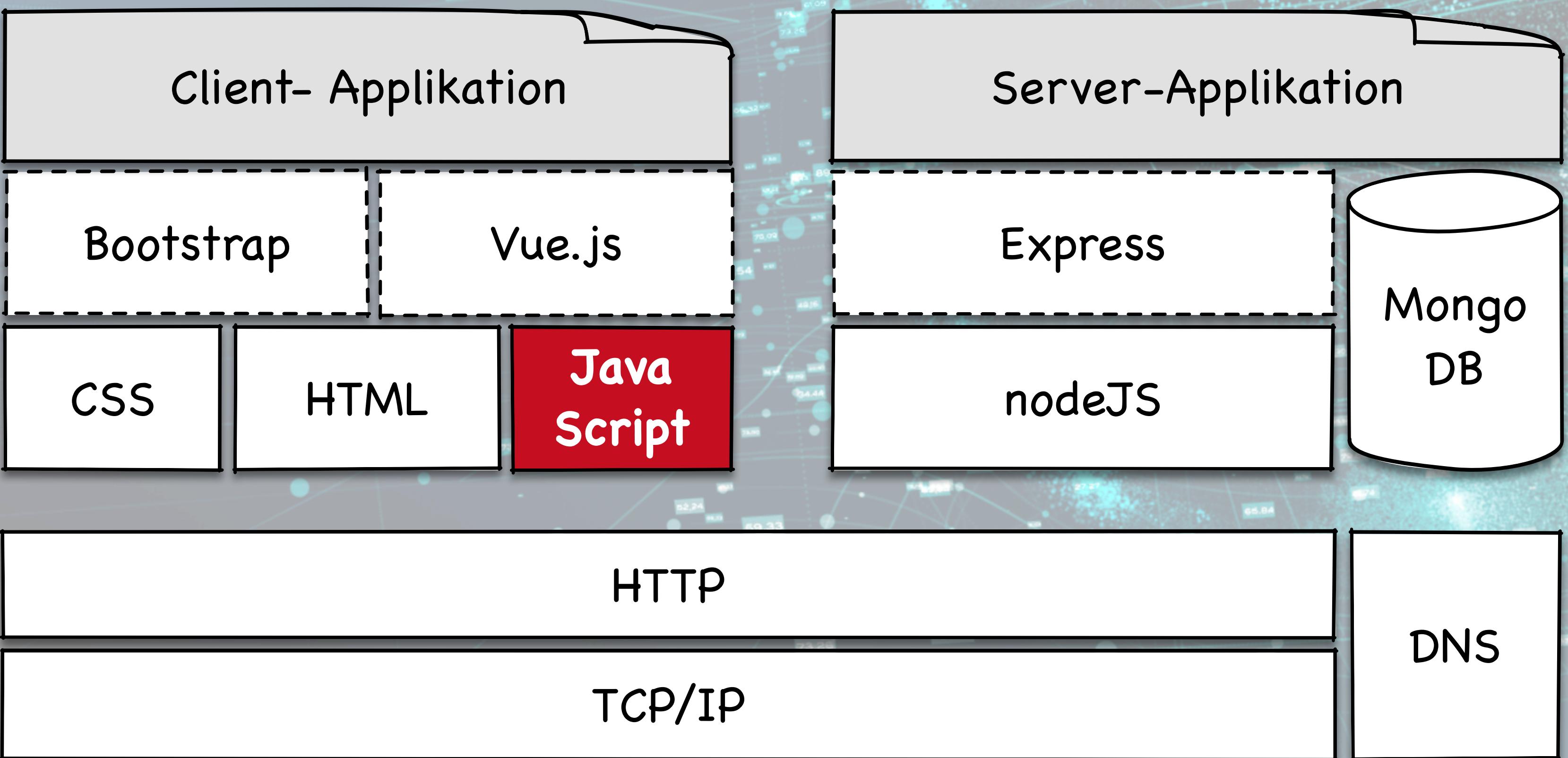


WEB TECHNOLOGIEN 2022

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 Komplilierungszeit 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

Warum?
Weil es auf ganz unterschiedlich Weise
gelesen wird (Smartwatch, PC, Mac ...), daher
Sollte Enthalten bleiben

↗ Just-in-Time Compilation

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

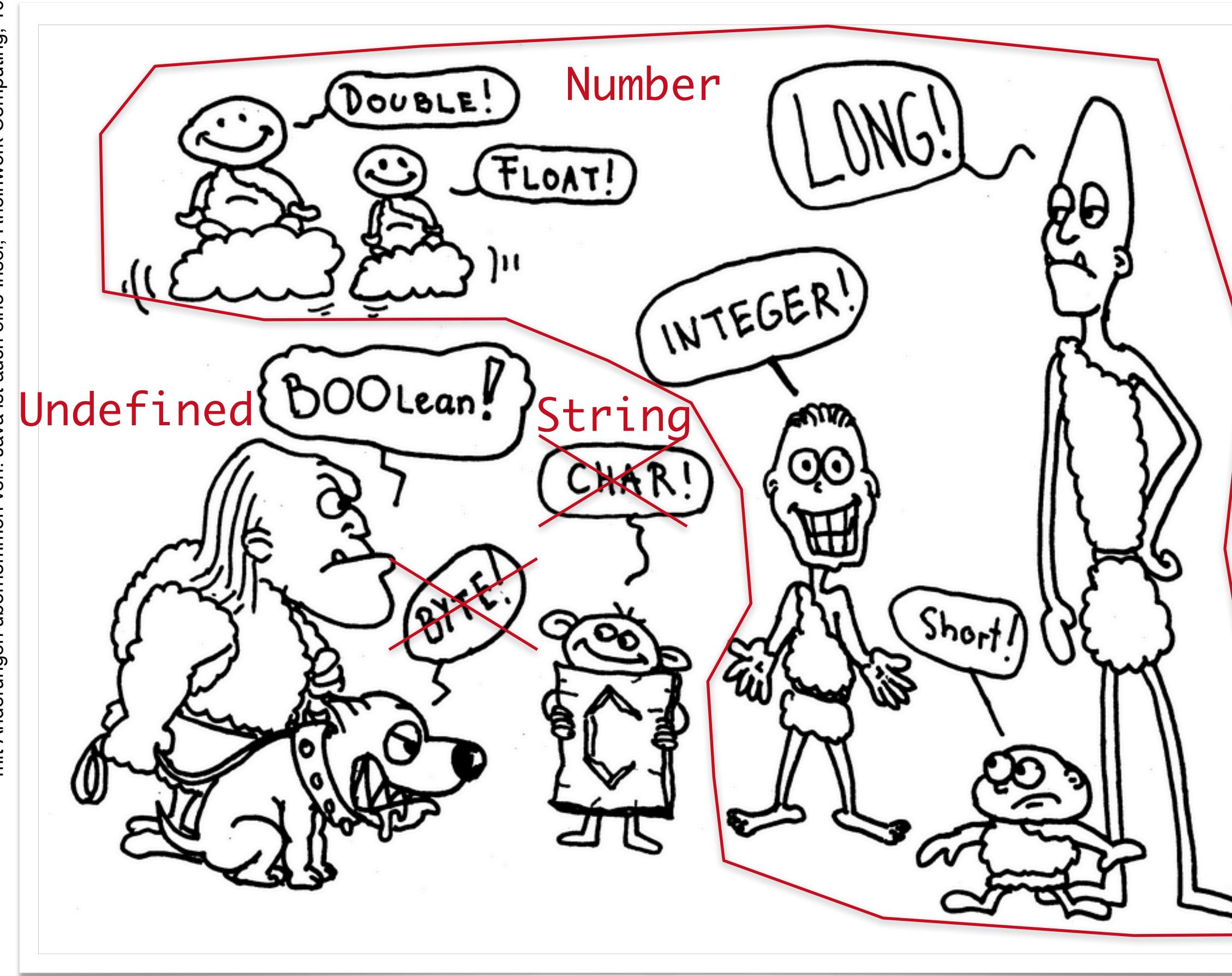
PROTOTYPISCHE OBJEKTORIENTIERUNG

- Umsetzung des objektorientierten Paradigmas basierend auf Prototypen, nicht Klassen

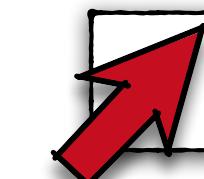
4.1 JAVASCRIPT-EINLEITUNG

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)



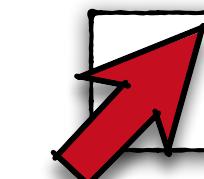
http://eloquentjavascript.net/01_values.html

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

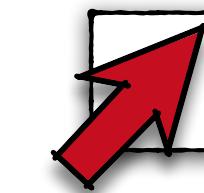


http://eloquentjavascript.net/01_values.html

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



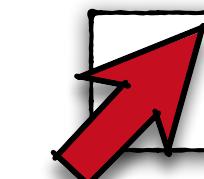
http://eloquentjavascript.net/01_values.html

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

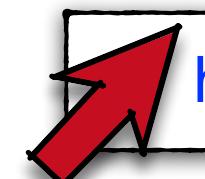
```
> 3>2
< true
> "Aachen"<"Zwickau"
< true
> "Peking"!="Beijing"
< true
> NaN==false
< false
> NaN==true
< false
> NaN==NaN
< false

> 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
```

Anweisung *Ausdruck* *Zeichenkette*

- 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: *(Da keine Berechnung eigentlich auch keine Response)*
 - Ausgabeanweisungen
 - Zuweisungsanweisung
 - Kontrollanweisung – *Schleifen*
 - Funktionsaufrufe
 - Ausdruck
- Anweisungen haben i.d.R keinen Wert
- Sequentielle Ausführung von Anweisungen ist ein Programm

4.3 PROGRAMMSTRUKTUREN VARIABLEN (I)

GETRENNTE DEKLARATION UND INITIALISIERUNG EINER VARIABLEN

```
> var ergebnis; Initialisierung
<- undefined
> ergebnis=5*5; Zuweisung
<- 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

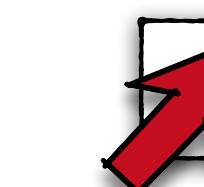
```
> var ergebnis=5*5;
<- undefined
> ergebnis=6*6;
<- 36
```

ÜBERSCHREIBEN EINES WERTES EINER VARIABLEN MIT EINEM WERT EINES ANDEREN DATENTYP

```
> var ergebnis=5*5; Schlecht, lass es nicht
<- undefined
> ergebnis="fünfundzwanzig";
<- "fünfundzwanzig"
```

Bei der Initialisierung von Variablen werden keine Datentypen festgelegt, dies passiert dynamisch

- 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
```

- mit `var` deklarierte Variablen sind auch außerhalb des Blocks sichtbar.
- mit `let` " " Sind nur innerhalb des Blocks sichtbar
- mit `const` " "
↳ mit `const` muss initialisiert und deklariert werden
können nicht im Block zweimal genutzt werden

- 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
```

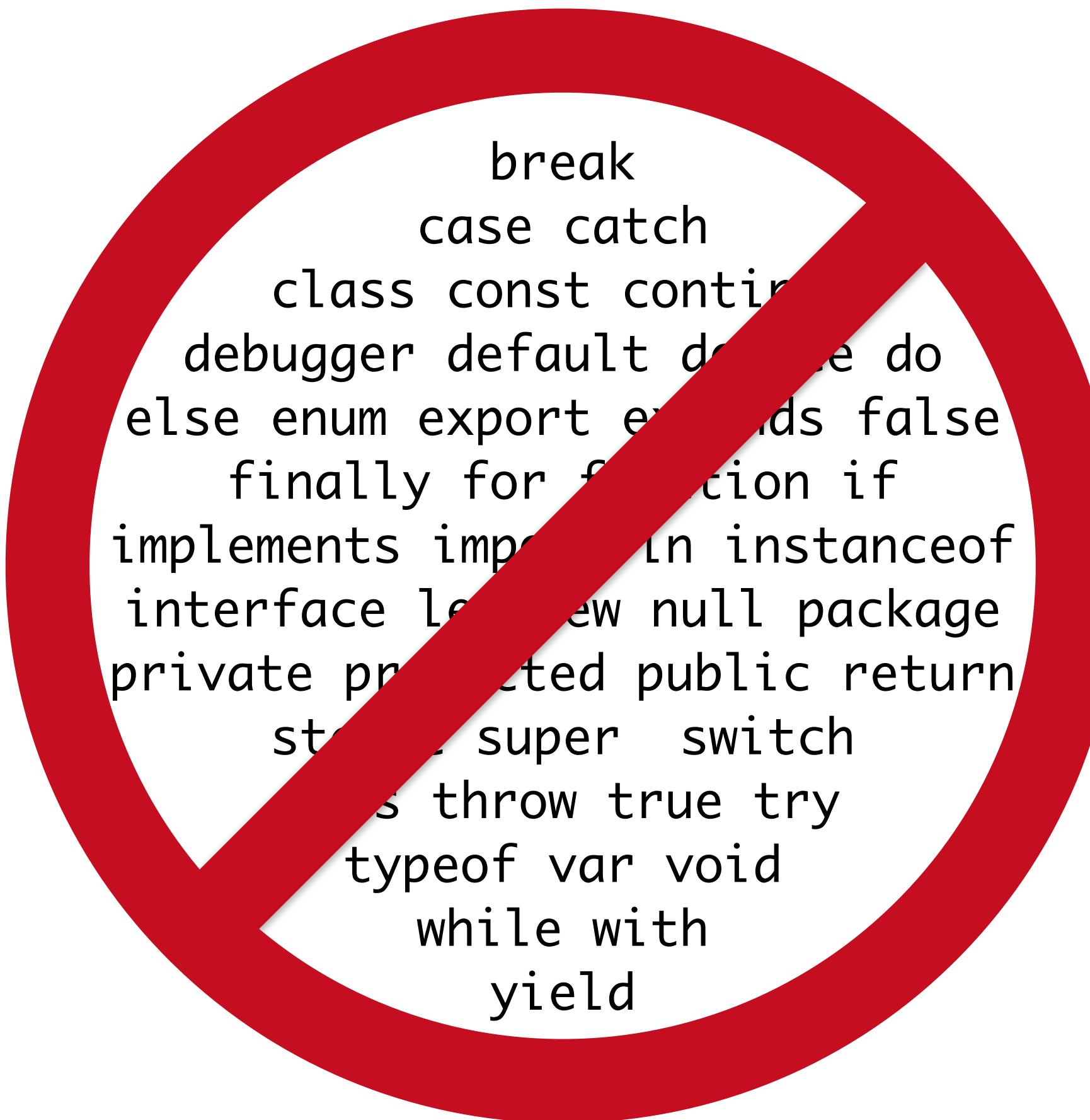
```
> {  
    let y=2;  
}  
console.log(y);  
✖ ▶ Uncaught ReferenceError: y is not defined  
    at <anonymous>:4:13
```

```
> 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

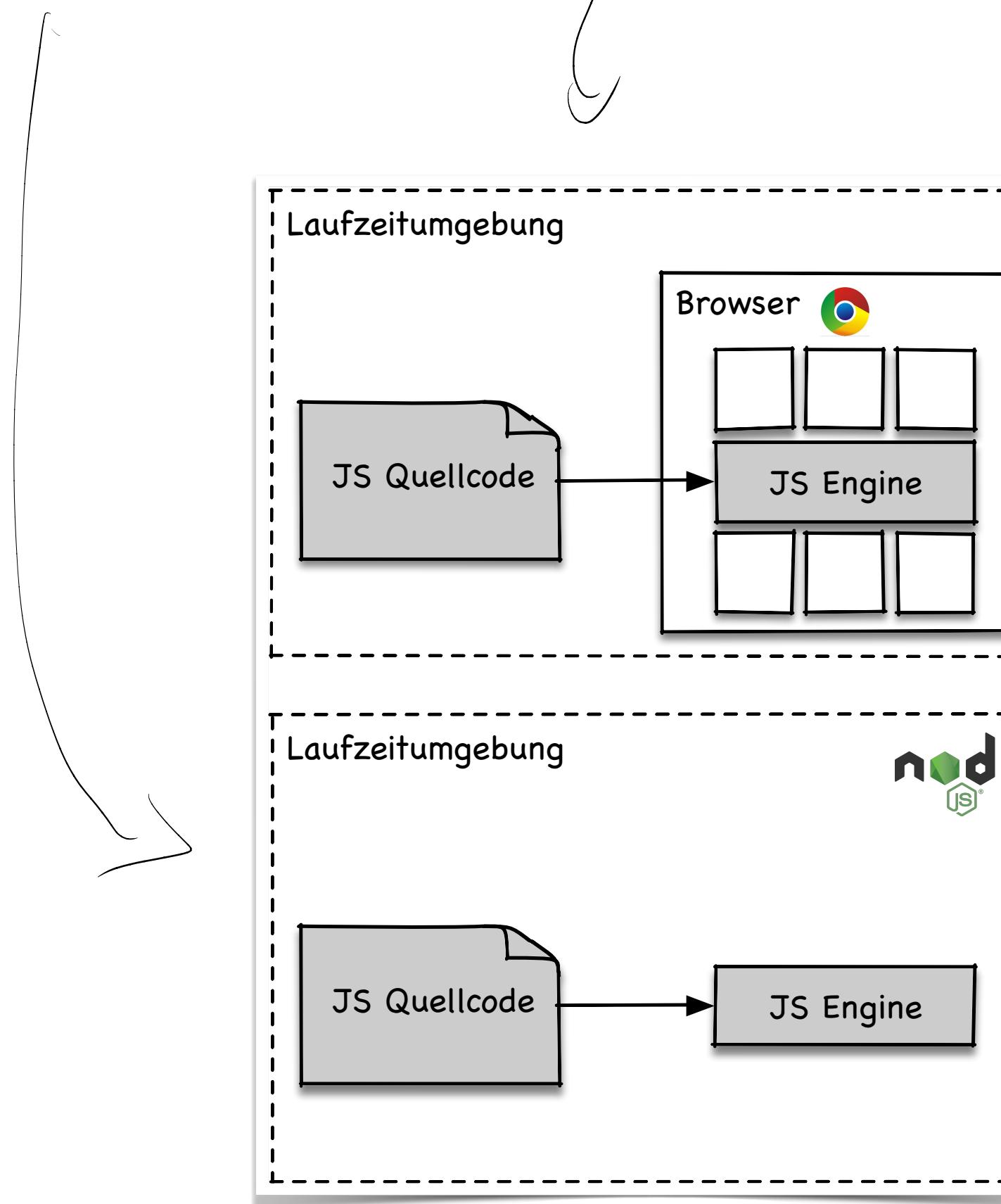


http://eloquentjavascript.net/02_program_structure.html

4.3 PROGRAMMSTRUKTUREN JAVASCRIPT-UMGEBUNGEN

Umggebungen von JS

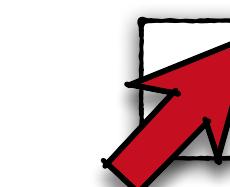
- 1. in Browser
- 2. in Server }



- 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

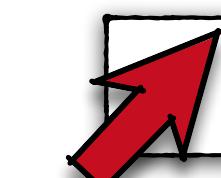
4.3 PROGRAMMSTRUKTUREN KONTROLLFLÜSSE (II)

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

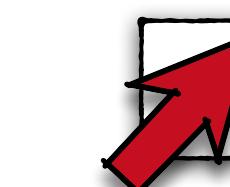
- Schleifen ermöglichen die wiederholte Ausführung einer bestimmten Sequenz von Anweisungen basierend auf einer Bedingung

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

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