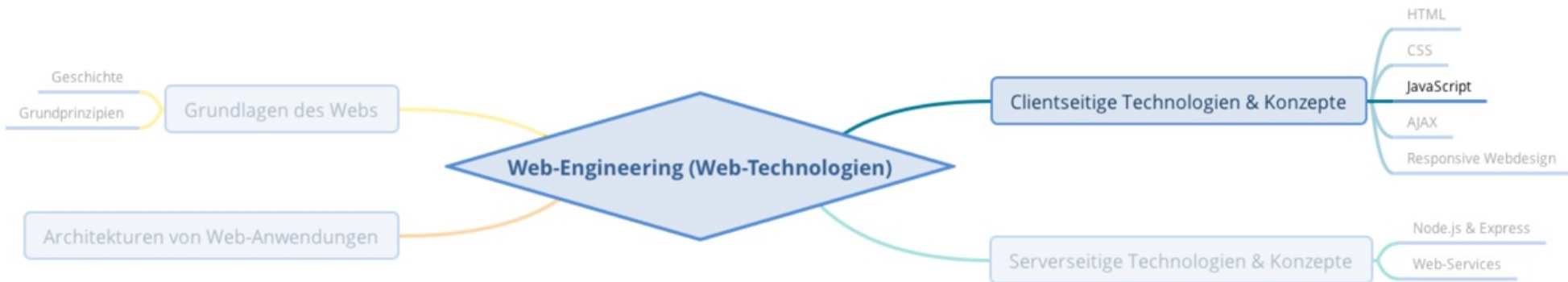




WEB- TECHNOLOGIEN

CLIENTSEITIGE TECHNOLOGIEN: JAVASCRIPT

THEMEN DER VERANSTALTUNG



LERNZIELE

1. Die grundlegenden Konstrukte von JavaScript kennen und anwenden können
2. Das Document Object Model (DOM) kennen und mit JavaScript manipulieren können

DIE WICHTIGSTEN WEB-TECHNOLOGIEN:



HTML → Inhalt
und Struktur



CSS → Darstellung



JavaScript →
Dynamik,
Verhalten

JAVASCRIPT

- JavaScript (häufig auch abgekürzt: *JS*) ist eine im Browser integrierte Programmiersprache → JavaScript-Code kann also auf dem Client ausgeführt werden
- Ursprünglich entwickelt, um einfaches Skripting im Browser zu ermöglichen
- Heute ist JavaScript nicht mehr auf den Browser beschränkt und ein fundamentaler Baustein für viele Anwendungsbereiche, z.B.:
 - Komplexe clientseitige Web-Frameworks (z.B. [Angular](#) , [React](#))
 - Einsatz auf dem Server (z.B. [Node.js](#) , [Rhino](#))
 - Entwicklung von Desktop-Anwendungen (z.B. mit [Electron](#))

JAVASCRIPT IM BROWSER

- In dieser Vorlesung konzentrieren wir uns auf die Anwendung von JavaScript im Browser
- Beispiele für Einsatzzwecke im Browser:
 - Dynamische Manipulation von Webseiten über das Document Object Model (DOM)
 - Reaktion auf und Verarbeitung von Ereignissen (z.B. Benutzereingaben)
 - Serverkommunikation im Hintergrund (z.B. Nachladen von Daten)
 - Verwendung von Web-APIs (z.B. [Geolocation](#) , [Canvas](#) , [Web Storage](#))

JAVASCRIPT: GESCHICHTE

1995



1995:

- Brendan Eich entwickelt für Netscape eine in den Browser eingebettete Skriptsprache mit dem Namen **LiveScript** (Netscape Navigator 2.0)
- Die Sprache sollte eigentlich auf Scheme (ein Lisp-Dialekt) basieren und für kleine Skripting-Aufgaben im Browser dienen

JAVASCRIPT: GESCHICHTE



1995

1995 (Fortsetzung):

- Durch eine Kooperation von Netscape und Sun wird die Sprache in **JavaScript** umbenannt und ihre Syntax eher an Java als an Scheme angelehnt - konzeptuell sind JavaScript und Java jedoch sehr unterschiedlich!

*"JavaScript" is as related to "Java" as "Carnival"
is to "Car".*

- Kyle Simpson

JAVASCRIPT: GESCHICHTE

1995

1996

1996:

- Aus Lizenzgründen implementiert Microsoft JavaScript selbst noch einmal und veröffentlicht es unter dem Namen **JScript** (Internet Explorer 3)
- Es entsteht der sogenannte Browserkrieg zwischen Netscape und Microsoft*
- JScript enthält zusätzlich Windows-spezifische Funktionalitäten

* Spoiler: Netscape unterliegt am Ende


JAVASCRIPT: GESCHICHTE



1996 (Fortsetzung):


- Netscape wendet sich an die [Ecma International](#) (früher: *European Computer Manufacturers Association*), um JavaScript zu standardisieren

JAVASCRIPT: GESCHICHTE



1995 1997
1996

1997:

- Die Standardisierungsbemühungen mündeten in der Veröffentlichung des Standards [ECMA-262](#) 
- Der offizielle Name dieser standardisierten Sprache lautet **ECMAScript**
- JavaScript und JScript sind Implementierungen dieses Standards (weiteres bekanntes Beispiel: ActionScript)

JAVASCRIPT: GESCHICHTE

1995 1997 1999

1996 1998

1998:

ECMAScript 2 wird veröffentlicht

1999:

ECMAScript 3 wird veröffentlicht

JAVASCRIPT: GESCHICHTE



2009:

ECMAScript 5 wird veröffentlicht
(ECMAScript 4 erscheint aufgrund von Streitigkeiten nie)

JAVASCRIPT: GESCHICHTE



2015:

ECMAScript 6 wird veröffentlicht:

- Auch genannt: ES6, ECMAScript 2015
- ECMAScript 2015 ist der offizielle Name, das zuständige Komitee ([TC39](#)) der Ecma wechselt ab hier auf jährliche Releases
- ES6 ist eine enorme Weiterentwicklung der Sprache

JAVASCRIPT: GESCHICHTE



2016:

ECMAScript 7 wird veröffentlicht
(auch genannt: ECMAScript 2016, ES7)


2017:

ECMAScript 8 wird veröffentlicht
(auch genannt: ECMAScript 2017, ES8)

...

MEHR ZU JAVASCRIPT-VERSIONEN:

- Übersicht zu JavaScript-Versionen auf [w3schools.com](https://www.w3schools.com) 
- Kompatibilitätstabellen für ECMAScript 

! Tipp: Bei caniuse.com  nach JavaScript-Funktionalitäten suchen (z.B. "arrow functions"), um Unterstützung in Browsern zu überprüfen

JAVASCRIPT-ENGINES

- Die Ausführung von JavaScript erfolgt über eine **JavaScript-Engine**
- Beispiele von Engines: [V8](#) , [SpiderMonkey](#) , [Rhino](#)
- Traditionell waren diese Engines *Interpreter*
- Heutige Engines arbeiten aus Performanzgründen jedoch typischerweise mit *Just-In-Time-Kompilierung* (JIT), d.h. Übersetzung in nativen Maschinencode

BEGRIFFSDEFINITIONEN*

Compiler

Programm, das Programme aus einer Programmiersprache A (Quellsprache) in eine Programmiersprache B (Zielsprache) übersetzt

Interpreter

Programm, welches ein Programm einer (anderen) Programmiersprache Anweisung für Anweisung analysiert und unmittelbar ausführt

Just-in-Time-Compiler (JIT-Compiler)

Compiler, der Programmteile in ein Programmiersprache - während der Laufzeit (*just-in-time*) - in ein Maschinenprogramm für eine spezielle Plattform übersetzt

* aus *Einführung in die Programmierung*

SPRACHEIGENSCHAFTEN

Multiparadigmatisch

JavaScript unterstützt verschiedene Programmierparadigmen:

- Prozedural
- Funktional
- Prototypenbasiert (auch: *objektbasiert*, *klassenlose Objektorientierung*)
- Ereignisgetrieben

SPRACHEIGENSCHAFTEN (2)

Dynamisch typisiert

Typüberprüfungen (z.B. bei Variablen) erfolgen erst zur Laufzeit (bei statisch typisierten Sprachen wie Java: zur Kompilierzeit)

- ! Es gibt Erweiterungen von JavaScript, die eine statische Typisierung ermöglichen (z.B. [TypeScript](#) , [Flow](#)).

EINBINDUNG VON JAVASCRIPT IN HTML

JavaScript-Code wird eingebunden über das `script`-Element:

1. **Internes JavaScript:** Direkt im HTML-Dokument, oder
2. **Externes JavaScript:** Durch Referenzieren einer externen Datei

INTERNES JAVASCRIPT

Der JavaScript-Code wird direkt als Inhalt des `script`-Elements
notiert

Beispiel:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Titel meiner Web-Seite</title>
    <meta charset="utf-8">
  </head>

  <body>
    <p id="p1"></p>
    <script>
      // Selektiert das Element mit der ID "p1" und
      // modifiziert dessen Inhalt
      document.getElementById("p1").innerHTML
        = "JavaScript was here!";
    </script>
  </body>
</html>
```

JavaScript was here!

EXTERNES JAVASCRIPT

Der JavaScript-Code wird in einer externen Datei (Endung .js) notiert und im `src`-Attribut des `script`-Elements referenziert

Beispiel:

script.js

```
// Selektiert das Element mit der ID "p1" und
// modifiziert dessen Inhalt
document.getElementById("p1").innerHTML
    = "JavaScript was here!";
```

seite.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Titel meiner Web-Seite</title>
    <meta charset="utf-8">
  </head>

  <body>
    <p id="p1"></p>
    <!-- Externe Datei einbinden -->
    <script src="script.js"></script>
  </body>
</html>
```

JavaScript was here!

DAS `script`-ELEMENT

- Kann sowohl im Kopfbereich (`head`) als auch im sichtbaren Bereich (`body`) des HTML-Dokuments notiert werden
- Es können mehrere `script`-Elemente in einem HTML-Dokument vorkommen
- Die Reihenfolge und auch Positionierung von `script`-Elementen ist relevant - dazu später mehr

INTERNES ODER EXTERNES JAVASCRIPT?

Argumentation ähnlich wie bei CSS:

- Internes JavaScript kann die Lesbarkeit des HTML-Codes verschlechtern
- Bei Verwendung von internem JavaScript wird Struktur (HTML) vom Verhalten (JS) nicht mehr sauber getrennt
- + Bei externem JavaScript wird diese Trennung sauber vollzogen
- + Externes JavaScript ist besser wiederverwendbar, da es in mehreren HTML-Dokumenten eingebunden werden kann

INTERNES ODER EXTERNES JAVASCRIPT?

(2)

- ➕ Dateien mit externem JavaScript können vom Browser gecached werden (→ ggf. schnelleres Laden von Seiten, die das externe JavaScript ebenfalls einbinden)
- ➕ Internes JavaScript kann trotzdem manchmal die bessere Wahl sein, z.B. aus Performanzgründen, wenn nur sehr wenig JavaScript-Code eingebunden werden soll

AUSFÜHRUNG VON JAVASCRIPT

Standardablauf:

1. Der Browser empfängt das angeforderte HTML-Dokument
2. Der Browser liest das Dokument "von oben nach unten" ein
(*Parsing*)
3. Trifft der Parser auf ein `script`-Element, so wird das Einlesen unterbrochen:
 - Internes JavaScript: Der JavaScript-Code wird sofort ausgeführt
 - Externes JavaScript: Der Browser lädt die externe Datei und führt den enthaltenen JavaScript-Code aus
4. Der Parser fährt fort mit dem Einlesen des HTML-Dokuments



ÜBERBLICK: SPRACHELEMENTE VON JAVASCRIPT

ALLGEMEINES

- JavaScript ist case-sensitiv
- Leerzeichen werden typischerweise ignoriert (außer in String-Literalen)
- Ein JavaScript-Programm besteht aus einer Menge von **Anweisungen** (*statements*)
- Eine Anweisung wird durch ein Semikolon abgeschlossen (eigentlich optional, aber nachdrücklich empfohlen!)
- Mehrere Anweisungen können zwischen geschweiften Klammern (`{ ... }`) als **Block** gruppiert werden

BEISPIEL

```
// Anweisungen werden mit Semikolons getrennt,  
// nUMber und number sind unterschiedliche Variablen  
const nUMber = 1337;  
const number = 42;  
// Leerzeichen werden ignoriert  
const      name =      "Douglas Crockford";  
  
if (number > 23) {  
    // Dies ist ein Block mit mehreren Anweisungen  
    console.log(number);  
    console.log(name);  
}
```

KOMMENTARE

Kommentare können einzeilig (eingeleitet mit `//`) oder mehrzeilig (zwischen `/*` ... `*/`) notiert werden

Beispiel:

```
// Einzeiliger Kommentar
const number = 42;
const name = "Douglas Crockford"; // Noch ein einzeiliger Kommentar

/*
  Kommentar, der sich über
  mehrere Zeilen erstreckt
*/
if (number > 23) {
  console.log(number);
}
```

AUSDRÜCKE

- Ein **Ausdruck** (*expression*) ist eine Auswertungsvorschrift, die bei der Ausführung eines Programmes ausgewertet wird und einen **Wert** liefert
- Ausdrücke können mit Hilfe von **Operatoren** verknüpft werden

💡 Analogie *Natürliche Sprache*:

- Anweisungen sind vergleichbar zu *Sätzen* einer Sprache
- Ausdrücke sind vergleichbar zu *Bestandteilen* eines Satzes (Satzglieder, Worte)
- Operatoren spielen eine vergleichbare Rolle wie *Konjunktionen* und *Interpunktion*

WERTE IN JAVASCRIPT

- Jeder Wert hat einen Typ
- JavaScript kennt folgende primitive Datentypen:
 - Wahrheitswerte (`boolean`)
 - Zahlen (`number`)
 - Zeichenketten (`string`)
 - Symbole (`symbol`) → Zum Erstellen eindeutiger Werte
 - `undefined`^{*}
 - `null`^{*}

^{*} Mehr dazu im Abschnitt "Variablen".

BOOLEAN-LITERALE

❓ *Literale?* → Syntaktische Elemente zur Darstellung von Werten

- Zwei Boolean-Literale: `true` und `false`
- Wahrheitswerte sind typischerweise Ergebnisse von:
 - *Logischen Ausdrücken*
 - *Relationalen Ausdrücken*

LOGISCHE AUSDRÜCKE

Logische Operatoren in JavaScript:

| Operator | Bedeutung |
|--------------------------------------|-------------------|
| <i>ausdruck1</i> && <i>ausdruck2</i> | Logisches UND |
| <i>ausdruck1</i> <i>ausdruck2</i> | Logisches ODER |
| ! <i>ausdruck</i> | Logische Negation |

! Weder die Operanden noch das Ergebnis eines logischen Ausdrucks müssen Wahrheitswerte sein → siehe *Typumwandlung*

RELATIONALE AUSDRÜCKE

Relationale Operatoren in JavaScript:

| Operator | Bedeutung |
|---------------------------------------|--|
| <i>ausdruck1</i> == <i>ausdruck2</i> | gleich (mit Typumwandlung, <i>loose equality</i>) |
| <i>ausdruck1</i> != <i>ausdruck2</i> | ungleich (mit Typumwandlung) |
| <i>ausdruck1</i> === <i>ausdruck2</i> | gleich (ohne Typumwandlung, <i>strict equality</i>) |
| <i>ausdruck1</i> !== <i>ausdruck2</i> | ungleich (ohne Typumwandlung) |
| <i>ausdruck1</i> < <i>ausdruck2</i> | kleiner als (mit Typumwandlung) |
| <i>ausdruck1</i> > <i>ausdruck2</i> | größer als (mit Typumwandlung) |
| <i>ausdruck1</i> <= <i>ausdruck2</i> | kleiner als oder gleich (mit Typumwandlung) |
| <i>ausdruck1</i> >= <i>ausdruck2</i> | größer als oder gleich (mit Typumwandlung) |

NUMBER-LITERALE

- Notiert als ganze Zahlen, Gleitkommazahlen oder in Exponentialschreibweise
- Beispiele: 42, 23.42, 4.2e3
- Zahlen sind intern immer 64-Bit-Gleitkommazahlen
- Weitere mögliche Werte für Zahlen:
 - NaN (= *not a number*): Wert entspricht keiner gültigen Zahl
 - Infinity/-Infinity: Gültiger Wertebereich überschritten/unterschritten
- Number-Werte sind typischerweise Ergebnis von *arithmetischen Ausdrücken*

ARITHMETISCHE AUSDRÜCKE

Arithmetische Operatoren in JavaScript:

| Operator | Bedeutung |
|---------------------------------------|---|
| <i>ausdruck1</i> + <i>ausdruck2</i> | Addition |
| <i>ausdruck1</i> - <i>ausdruck2</i> | Subtraktion |
| <i>ausdruck1</i> * <i>ausdruck2</i> | Multiplikation |
| <i>ausdruck1</i> / <i>ausdruck2</i> | Division |
| <i>ausdruck1</i> % <i>ausdruck2</i> | Modulo (Rest bei ganzzahliger Division) |
| <i>++ausdruck</i> , <i>ausdruck++</i> | Präinkrement, Postinkrement |
| <i>--ausdruck</i> , <i>ausdruck--</i> | Prädekrement, Postdekrement |
| <i>-ausdruck</i> | Unäre Negation |

STRING-LITERALE

- Werden mit einfachen oder doppelten Anführungszeichen geschrieben
- Beispiele: "Web", 'Web '
- Über den +-Operator können Strings konkateniert werden, z.B.:

```
const name = "Technologien";  
const titel = "Web" + " " + name; // --> ergibt "Web Technologien"
```

TEMPLATE-LITERALE

- **Template-Literale** (*template literals*, Template = "Vorlage") bieten eine mächtige Alternative zur Erzeugung von Strings
- Statt einfacher oder doppelter Anführungszeichen wird bei Template-Literalen das *Gravis-Zeichen* ` (engl. *backtick*) verwendet
- Zudem können Template-Literale Platzhalter enthalten:
 - Form: `${ ausdruck }`
 - *ausdruck* ist ein Ausdruck, der dynamisch ausgewertet wird (*interpolation*)

TEMPLATE-LITERALE: BEISPIEL

Gewünschte Ausgabe:

You don't know JavaScript von Kyle Simpson
erschienen vor 3 Jahren

Code:

```
const titel = "You don't know JavaScript";
const autor = "Kyle Simpson";
const jahr = 2015;
const aktJahr = (new Date()).getFullYear(); // aktuelles Jahr ermitteln

// Ausgabe mit normaler String-Konkatentation aufbauen
const ausgabe1 = titel + " von " + autor + "\nerschienen vor "
                  + (aktJahr - jahr) + " Jahren";

// Ausgabe mit Template-Literal aufbauen
const ausgabe2 = `${titel} von ${autor}
erschienen vor ${aktJahr - jahr} Jahren`;

// In beiden Fällen wird die gleiche Ausgabe produziert
console.log(ausgabe1);
console.log(ausgabe2);
```

AUTOMATISCHE TYPUMWANDLUNG

- Trifft JavaScript auf einen unerwarteten Wert (z.B. bei einem Vergleich oder einem Funktionsaufruf), so wird versucht, diesen Wert zu konvertieren (*implicit coercion*)
- Die Regeln, nach denen diese Konvertierung vorgeht, können verwirrende Ergebnisse produzieren, z.B.:

```
"23" == 23
// --> true. Erklärung: "23" wird zu 23 konvertiert, 23==23 ergibt true

5 < "15"
// --> true. Erklärung: "15" wird zu 15 konvertiert, 5<15 ergibt true

"5" < "15"
// --> false. Erklärung: Keine Typkonvertierung, da beide Typen gleich sind.
// Daher werden die Strings lexikografisch verglichen.

false == ""
// --> true. Erklärung: false und "" werden zu 0 konvertiert, 0==0 ergibt true
```

AUTOMATISCHE TYPUMWANDLUNG (2)

Mögliche Strategien:

1. Typumwandlung vermeiden und `===` statt `==` (bzw. `!==` statt `!=`) verwenden (`===` und `!==` vergleichen ohne Typumwandlung):

```
"23" === 23 // --> false  
false === "" // --> false
```

2. Regeln der automatischen Typumwandlung [anschauen, verstehen und bewusst anwenden](#) 

TYPUMWANDLUNG BEI BOOLEAN-WERTEN

- Alle "nicht echten Werte" werden von JavaScript als `false` interpretiert ("*falsy*"): `NaN`, `0`, `-0`, `null`, `undefined`, `" "`
- Alle anderen Werte werden als `true` interpretiert ("*truthy*")

Beispiele (`Boolean(x)` konvertiert `x` in einen Boolean-Wert):

```
Boolean(0) // --> false
Boolean(23 / "Horst") // ergibt NaN --> false
Boolean(42) // --> true
Boolean("0") // --> true
Boolean("false") // --> true

true && "Dog"
// --> "Dog". Erklärung: ausdr1 && ausdr2 ergibt den Wert von ausdr1, wenn ausdr1
// false ergibt. Ansonsten ergibt der Ausdruck den Wert von ausdr2.
"Cat" || "Dog"
// --> "Cat". Erklärung: ausdr1 || ausdr2 ergibt den Wert von ausdr1, wenn ausdr1
// true ergibt. Ansonsten ergibt der Ausdruck den Wert von ausdr2.
```

VARIABLEN

- Variable = "Container" für Werte
- JavaScript bietet drei Schlüsselworte zur Deklaration von Variablen:
 - `const`: Deklaration einer Konstanten (darf nach der Initialisierung nicht mehr geändert werden)
 - `let`: Deklaration einer Variablen mit Blocksichtbarkeit^{*}
 - `var`: Deklaration einer Variablen mit Funktionssichtbarkeit^{*}
- Eine deklarierte, aber noch nicht initialisierte Variable hat den Wert `undefined`

^{*} Auf den Unterschied der Sichtbarkeiten gehen wir später genauer ein.

VARIABLEN: BEISPIELE

```
let number; // Deklaration der Variable "number", Wert: undefined
number = 42; // Initialisierung der Variable "number"

let name = "john"; // Deklaration und Initialisierung in einer Anweisung

/* Deklaration (farbe, alter, durchschnitt) und Initialisierung (farbe, alter)
   mehrerer Variablen in einer Anweisung */
let farbe = "rot", alter = 42, durchschnitt;

const PI = 3.14; // Deklaration der Konstante PI
PI = 2.34; // Fehler! Konstante darf nicht geändert werden

let alter = null; // "null" bedeutet "kein Wert"
```

VARIABLEN UND DATENTYPEN

- Variablen haben in JavaScript keinen Typ - lediglich *Werte* haben einen Typ
- Aus diesem Grund können ein und derselben Variablen Werte unterschiedlichen Typs zugewiesen werden:

```
// Deklaration und Initialisierung der Variable "number"  
// mit einem Number-Wert  
let number = 42;  
// Zuweisung eines String-Wertes  
number = "forty-two";  
// Zuweisung eines Boolean-Wertes  
number = true;
```

VARIABLEN UND DATENTYPEN (2)

- Mit dem `typeof`-Operator kann der Typ eines Wertes ermittelt werden:

```
let number = 42;
console.log(typeof(number)); // Ausgabe: number

let bool = true;
console.log(typeof(bool)); // Ausgabe: boolean

let name = "John";
console.log(typeof(name)); // Ausgabe: string

let color;
console.log(typeof(color)); // Ausgabe: undefined
```


FUNKTIONEN

- Eine **Funktion** ist ein benannter Block von Anweisungen
- Der Name der Funktion kann verwendet werden, um die Funktion aufzurufen
- Optional kann eine Funktion Parameter definieren, die beim Aufruf als Argumente übergeben werden können
- Optional kann eine Funktion einen Wert als Rückgabe zurückliefern

FUNKTIONEN: BEISPIEL

```
// Definition einer Funktion mit Namen "logDate"
function logDate() {
    console.log(Date());
}

logDate(); // Aufruf der logDate-Funktion

// Definition einer Funktion mit Namen "greet" und Parameter "name"
function greet(name) {
    return "Hallo " + name;
}

// Aufruf der greet-Funktion mit Argument, Ausgabe: "Hallo WEB1"
console.log(greet("WEB1"));
```

FUNKTIONEN (2)

- Funktionen können Variablen zugewiesen werden
- Funktionen können Funktionen als Argumente übergeben werden

```
// Zuweisung einer anonymen Funktion zur Variable "greeter"
let greeter = function(name) {
    return "Hallo " + name;
}

// Definition einer Funktion, die im Parameter "greeterFunction"
// eine weitere Funktion erwartet und diese dann aufruft
function logDateWithGreeting(greeterFunction, name) {
    console.log(greeterFunction(name) + " " + Date());
}

// Aufruf der Funktion "logDateWithGreeting" mit der Funktion
// "greeter" als Argument
logDateWithGreeting(greeter, "WEB1");
```

FUNKTIONEN (3)

- Die für eine Funktion definierten Parameter sind nicht zwingend:
Es können auch weniger oder mehr Argumente übergeben werden
- Fehlende Argumente erhalten in der Funktion den Wert `undefined`
- Zum Zugriff auf zusätzliche Argumente gibt es in der Funktion das Objekt `arguments`

```
function printMessage(msg) {  
    console.log(msg);  
}  
printMessage(); // Ausgabe: "undefined"  
  
function printMessages() {  
    for (let i=0; i < arguments.length; i++) {  
        console.log(arguments[i]);  
    }  
}  
// Ausgabe:  
// "Message 1"  
// "Message 2"  
printMessages("Message 1", "Message 2");
```

BEDINGTE ANWEISUNGEN UND SCHLEIFEN

Die Syntax für bedingte Anweisungen und Schleifen ist vergleichbar zu den entsprechenden Konstrukten in Java

Verzweigung

```
let monat = new Date().getMonth();

if (monat >= 0 && monat <= 2) {
    console.log("Erstes Quartal");
} else if (monat >= 3 && monat <= 5) {
    console.log("Zweites Quartal");
} else {
    console.log("Zweite Jahreshälfte")
}
```

Mehrfachverzweigung

```
switch(new Date().getMonth()) {
    case 0:
        console.log("Januar");
        break;
    case 11:
        console.log("Dezember");
        break;
    default:
        console.log("Nicht Januar oder Dezember");
}
```

BEDINGTE ANWEISUNGEN UND SCHLEIFEN (2)

While-Schleife

```
let i = 0;
while (i < 5) {
  console.log("Nummer " + i);
  i++;
}
```

Do-While-Schleife

```
let i = 0;
do {
  console.log("Nummer " + i);
  i++;
} while (i < 5);
```

For-Schleife

```
let i = 0;
for (let i = 0; i < 5; i++) {
  console.log("Nummer " + i);
}
```

SICHTBARKEIT VON VARIABLEN

- JavaScript kennt drei Sichtbarkeitsbereiche:
 1. Globale Sichtbarkeit (*global scope*)
 2. Sichtbarkeit auf Funktionsebene (*function scope*)
 3. Sichtbarkeit auf Blockebene (*block scope*)
- Mit `var` deklarierte Variablen sind sichtbar innerhalb der umgebenden Funktion
- Mit `let` oder `const` deklarierte Variablen sind sichtbar innerhalb des umgebenden Blocks
- Bei Deklaration auf oberster Ebene (d.h. nicht in einer Funktion oder einem Block) sind alle drei Varianten global sichtbar

SICHTBARKEIT: BEISPIELE

```
var a = 1;
let b = 2;

function scopeTest() {
  var a = 11;
  let b = 12;
  if (true) {
    var a = 111;
    let b = 122;

    console.log(a, b); // Ausgabe: 111, 122
  }

  console.log(a, b); // Ausgabe: 111, 12
}

scopeTest();
console.log(a, b); // Ausgabe: 1, 2
```


OBJEKTE

- Neben den uns schon bekannten primitiven Datentypen gibt es in JavaScript noch einen komplexen Datentyp: `object`
- Ein Objekt in JavaScript ist eine Liste von Eigenschaften (*properties*) und zugehörigen Werten (*values*)

OBJEKTE: ZUGRIFF

Der Zugriff auf die Eigenschaften eines Objektes erfolgt per Punktnotation (`objekt.eigenschaft`) oder per Indexnotation (`objekt["eigenschaft"]`)

Beispiel:

```
// Erzeugung eines Objektes als Objektliteral
let film = {
  titel: "Pi",
  regisseur: "Darren Aronofsky",
  jahr: 1998
}

console.log(film.titel); // Ausgabe: "Pi"
console.log(film["titel"]); // Ausgabe: "Pi"
```

METHODEN

Methoden des Objekts sind Eigenschaften, deren Wert eine Funktion ist

Beispiel:

```
// Erzeugung eines Objektes als Objektliteral
let film = {
  titel: "Pi",
  regisseur: "Darren Aronofsky",
  jahr: 1998,
  berechneAlter: function() {
    return new Date().getFullYear() - this.jahr;
  }
}

console.log(film.berechneAlter()); // Ausgabe: "20"
console.log(film["berechneAlter"]()); // Ausgabe: "20"
```

OBJEKTERZEUGUNG

JavaScript kennt drei Wege zur Objekterzeugung:

1. Objektliterale
2. Konstruktorfunktionen
3. Klassennotation

OBJEKTITERALE

- Erzeugung eines einzelnen Objektes
- War das Vorbild für die JavaScript Object Notation ([JSON](#) )

Beispiel:

```
// Erzeugung eines Objektes als Objektliteral
let film = {
  titel: "Pi",
  regisseur: "Darren Aronofsky",
  jahr: 1998,
  berechneAlter: function() {
    return new Date().getFullYear() - this.jahr;
  }
}
```

KONSTRUKTORFUNKTIONEN

- Zur Erzeugung beliebig vieler Instanzen eines Objektes
- Normale Funktion (Konvention: Ersten Buchstaben des Namens groß schreiben)
- Durch Aufruf der Funktion mit dem Schlüsselwort `new` wird ein neues Objekt erzeugt

Beispiel:

```
function Film(titel, regisseur, jahr) {  
    this.titel = titel;  
    this.regisseur = regisseur;  
    this.jahr = jahr;  
    this.berechneAlter = function() {  
        return new Date().getFullYear() - this.jahr;  
    };  
}  
  
let film1 = new Film("Pi", "Darren Aronofsky", 1998);  
let film2 = new Film("Videodrome", "David Cronenberg", 1983);  
  
console.log(film1.berechneAlter()); // Ausgabe: "20"  
console.log(film2.berechneAlter()); // Ausgabe: "35"
```

KLASSEN

- Definition einer "Klasse" über das Schlüsselwort `class` sowie eines Konstruktors mittels `constructor`
- Keine echten Klassen wie z.B. in Java - lediglich eine andere Schreibweise (intern immer noch Konstruktorfunktionen)
- Bei Methodendefinition darf das Schlüsselwort `function` weggelassen werden

KLASSEN: BEISPIEL

```
// Definition der "Klasse"
class Film {
    // Konstruktor - nur einmal pro Klassendefinition erlaubt
    constructor(titel, regisseur, jahr) {
        this.titel = titel;
        this.regisseur = regisseur;
        this.jahr = jahr;
    }

    // Methode ohne "function"-Schlüsselwort (seit ES6)
    berechneAlter() {
        return new Date().getFullYear() - this.jahr;
    };
}

let film1 = new Film("Pi", "Darren Aronofsky", 1998);
let film2 = new Film("Videodrome", "David Cronenberg", 1983);

console.log(film1.berechneAlter()); // Ausgabe: "20"
console.log(film2.berechneAlter()); // Ausgabe: "35"
```


MODIFIKATION VON OBJEKTEN

- Einzelnen Objekten (d.h. Instanzen!) können jederzeit Eigenschaften (Attribute, Methoden) hinzugefügt oder entfernt werden
- Das Hinzufügen geschieht durch einfache Zuweisung
- Für das Entfernen gibt es das Schlüsselwort `delete`

MODIFIKATION VON OBJEKTEN:BEISPIEL

```
/* Konstruktorfunktion (alternative Schreibweise mit anonymer Funktion) */
let Film = function(titel, regisseur, jahr) {
  this.titel = titel;
  this.regisseur = regisseur;
  this.jahr = jahr;
}

let film1 = new Film("Pi", "Darren Aronofsky", 1998);
let film2 = new Film("Videodrome", "David Cronenberg", 1983);

// neues Attribut zu "film1" hinzufügen
film1.dauer = 130;
// neue Funktion zu "film2" hinzufügen
film2.berechneAlter = function() {
  return new Date().getFullYear() - this.jahr;
}

// nur film1 hat die Methode "berechneAlter"
console.log(film1.dauer); // Ausgabe: 130
console.log(film2.dauer); // Ausgabe: undefined

// nur film2 hat die Methode "berechneAlter"
console.log(film1.berechneAlter()); // TypeError: film1.berechneAlter is not a function
console.log(film2.berechneAlter()); // Ausgabe: 35

// Attribut "titel" inklusive Wert entfernen
delete film1.titel;
console.log(film1.titel); // Ausgabe: undefined
```

STANDARD OBJEKTE

JavaScript bietet einige Standardobjekte mit nützlichen Funktionen,
z.B.:

- Wrapper-Objekte für primitive Datentypen: `String`, `Number`, `Boolean`
- `Function` (Alle Funktionen sind Objekte!)
- `RegExp` für reguläre Ausdrücke
- `Date` zum Umgang mit Datum und Uhrzeit
- `Array`

STANDARD OBJEKTE

JavaScript bietet einige Standardobjekte mit nützlichen Funktionen,
z.B.:

- Wrapper-Objekte für primitive Datentypen: `String`, `Number`, `Boolean`
- `Function` (Alle Funktionen sind Objekte!)
- `RegExp` für reguläre Ausdrücke
- `Date` zum Umgang mit Datum und Uhrzeit
- `Array`

STANDARD OBJEKTE: `String`

- Objektvariante zum primitiven Datentyp `String`
- Beispiele für Eigenschaften und Methoden:

| <code>length</code> | Länge der Zeichenkette |
|--|--|
| <code>slice(start, end)</code> | Extrahiert den Teil vom Startindex <code>start</code> bis zum Endindex <code>end</code> (exklusive) und gibt ihn als neue Zeichenkette zurück. |
| <code>charAt(position)</code> | Ermittelt das Zeichen an Position <code>position</code> . |
| <code>replace(string1, string2)</code> | Ersetzt das erste Vorkommen von <code>string1</code> durch <code>string2</code> . |
| <code>toLowerCase()</code> | Konvertiert alle Zeichen in Kleinbuchstaben. |

STANDARD OBJEKTE: `Number`

- Objektvariante zum primitiven Datentyp `number`
- Beispiele für Methoden:

`toString(basis)` Gibt eine Zeichenkette zurück, die die Zahl zur Basis `basis` repräsentiert.

`toFixed(digits)` Konvertiert die Zahl in Gleitkommanotation, `digits` gibt die Anzahl von Nachkommastellen an.

String, Number, Boolean

- ! Aus Effizienzgründen sollten nicht die jeweiligen Objektvarianten zur Erstellung von Zeichenketten, Zahlen und Wahrheitswerten verwendet werden.
- Dies ist in der Praxis auch gar nicht nötig: Eigenschaften und Methoden können direkt auf den primitiven Datentypen aufgerufen werden (automatische Typumwandlung!).

BEISPIELE: OBJEKTVARIANTEN

```
let name = "Starbuck";
let name2 = new String("Apollo"); // nicht empfohlen

/* Eigenschaften und Methoden können direkt auf dem primitiven Datentyp aufgerufen werden -
   JS konvertiert diese dazu automatisch in die Objektvariante */
console.log(name.length); // Ausgabe: 8
console.log(name.toUpperCase()); // Ausgabe: STARBUCK
console.log(name.slice(0, 4)); // Ausgabe: Star

let number = 42;
console.log(number.toString(2)); // Ausgabe: 101010
console.log(number.toFixed(3)); // Ausgabe: 42.000
```


ARRAYS

- Geordnete Sammlung mehrerer Werte
- Die einzelnen Werte (*Elemente*) im Array sind nummeriert (*Index*), startend mit dem Index 0
- Elemente eines Arrays können beliebige Typen haben
- Arrays können per Literalschreibweise oder Konstruktorfunktion des Array-Objektes erzeugt werden:

```
// Array mit Literalschreibweise erzeugen (üblicher)
let array = ["Starbuck", 42, false, "Apollo"];
// Array mit Konstruktorfunktion erzeugen
let array2 = new Array("Starbuck", 42, false, "Apollo");

// Kein Unterschied: In beiden Fällen wird ein Object erzeugt
console.log(typeof(array), typeof(array2)); // Ausgabe: object object

// Zugriff auf Array-Elemente per Index
console.log(array[0], array[3], array[1000]) // Ausgabe: Starbuck Apollo undefined
```

ARRAYS: LÄNGE

- Die Eigenschaft `length` liefert die Länge eines Arrays
- Die Größe des Arrays ist nicht fix: In einem Array können jederzeit weitere Elemente hinzugefügt oder entfernt werden^{*}

```
let array = ["Starbuck", 42, false, "Apollo"];
console.log(array.length); // Ausgabe: 4

/* Element an Index 7 hinzufügen - Array wird dynamisch vergrößert
   und leere Einträge bekommen den Wert "undefined" */
array[7] = "Helo";
// Element mit Index 2 löschen (Wert wird auf "undefined" gesetzt)
delete array[2];

console.log(array.length); // Ausgabe: 8
console.log(array[2], array[6]); // Ausgabe undefined undefined
// Ausgabe des gesamten Array per "toString"
console.log(array.toString()); // Ausgabe: Starbuck,42,,Apollo,,,Helo
```

^{*} Im Vergleich zu Java verhalten sich JavaScript-Arrays also eher wie die Java-Collections und nicht wie die Java-Arrays.

ARRAYS: METHODEN

Beispiele für Methoden von Arrays:

| | |
|---|---|
| <code>pop()</code> | Entfernt das letzte Element des Arrays und gibt es zurück |
| <code>push(element)</code> | Fügt das Element <code>element</code> am Ende des Arrays hinzu und gibt die neue Länge des Arrays zurück. |
| <code>splice(index, deleteNr, element₁, ..., element_n)</code> | Entfernt <code>deleteNr</code> viele Elemente an Position <code>index</code> und fügt ab dieser Stelle die neuen Elemente <code>element₁-element_n</code> ein. |
| <code>sort(compareFunction)</code> | Sortiert das Array. Optional kann eine Funktion <code>compareFunction</code> angegeben werden, die eine Sortierung definiert. |
| <code>join(separator)</code> | Erzeugt eine Zeichenkette aus dem Array, wobei die Elemente mit dem gegebenen <code>separator</code> getrennt werden (falls nicht gegeben: ","). |

ARRAY-METHODEN: BEISPIELE

```
let array = ["Starbuck", 42, false, "Apollo"];
console.log(array.length); // Ausgabe: 4

console.log(array.pop()); // Ausgabe: Apollo
console.log(array.push("Boomer")); // Ausgabe: 4

array.splice(1, 2, "Helo");
console.log(array.join(" - ")); // Ausgabe: Starbuck - Helo - Boomer

let array2 = [42, 23, 256, 7];
// Standard ist lexikografische Sortierung
array2.sort();
console.log(array2.toString()); // Ausgabe: 23,256,42,7
// Funktion mitgeben, die numerisch sortiert
array2.sort(function(x, y) {
    /* Erwartete Rückgabe: negativer Wert bei x<y,
     * 0 bei x=y, positiver Wert bei x>y
     */
    return x-y;
});
console.log(array2.toString()); // Ausgabe: 7,23,42,256
```

ARRAYS DURCHLAUFEN

Verschiedene Möglichkeiten, Arrays zu durchlaufen:

Zählschleife

```
let array = ["Starbuck", "Boomer", "Apollo"];
for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}
```

for-of-Schleife

```
let array = ["Starbuck", "Boomer", "Apollo"];
for (let elem of array) {
  console.log(elem);
}
```

Funktional über **forEach**-Methode

```
let array = ["Starbuck", "Boomer", "Apollo"];
array.forEach(function(elem) {
  console.log(elem);
});
```

PROTOTYPEN

- Erinnerung: JavaScript ist **prototypenbasiert** (auch: *objektbasiert*, *klassenlose Objektorientierung*)
- Jedem Objekt in JavaScript ist ein weiteres Objekt zugeordnet, welches sein **Prototyp** ist
- Das Objekt erbt Eigenschaften (d.h. Attribute und Methoden) von seinem Prototypen
- Auch der Prototyp hat wiederum einen Prototypen usw. - es entsteht eine **Prototypenkette**

PROTOTYPENKETTE

- Das "oberste" Objekt der Prototypenkette ist `Object.prototype`, dessen Prototyp `null` ist
- Mittels Objektliteral erzeugte Objekte haben standardmäßig `Object.prototype` als Prototyp (bietet einige Standardeigenschaften)
- Mittels Konstrukturfunktion und `new` erzeugte Objekte erhalten den Wert der Eigenschaft `prototype` der Konstrukturfunktion als Prototyp

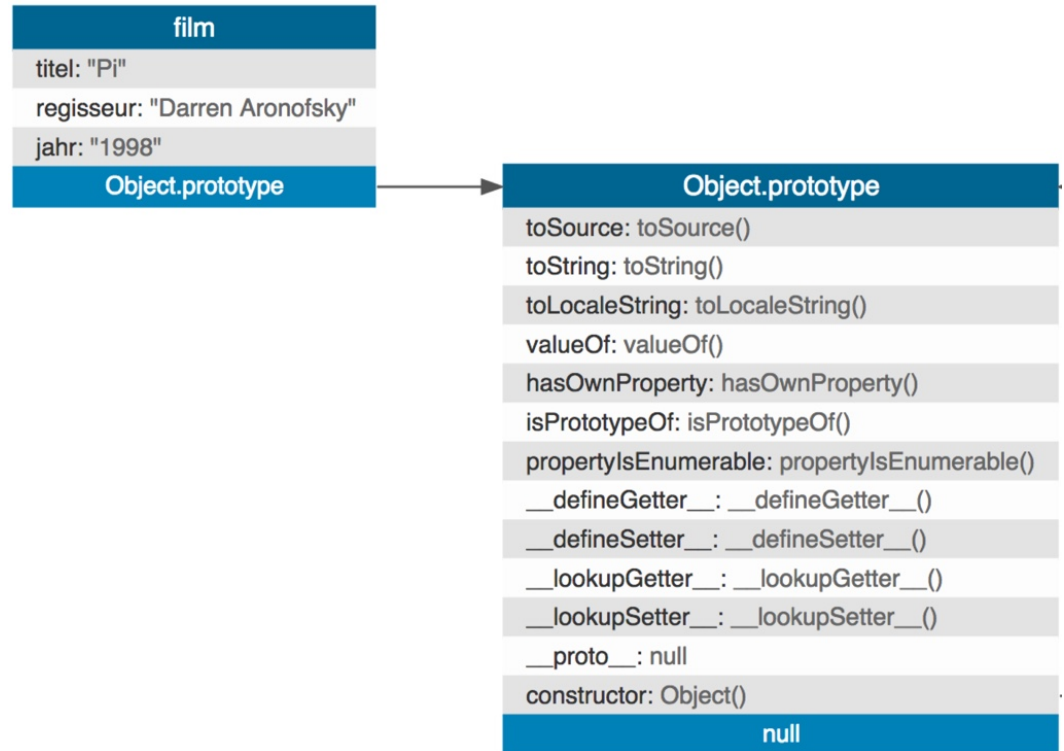
PROTOTYPENKETTE: BEISPIELE

Beispiel 1: Objekterzeugung mittels Objektliteral

JavaScript:

```
let film = {  
  titel: "Pi",  
  regisseur: "Darren Aronofsky",  
  jahr: 1998  
}
```

Prototypenkette:



Visualisierung erzeugt mit [Object Playground](#)

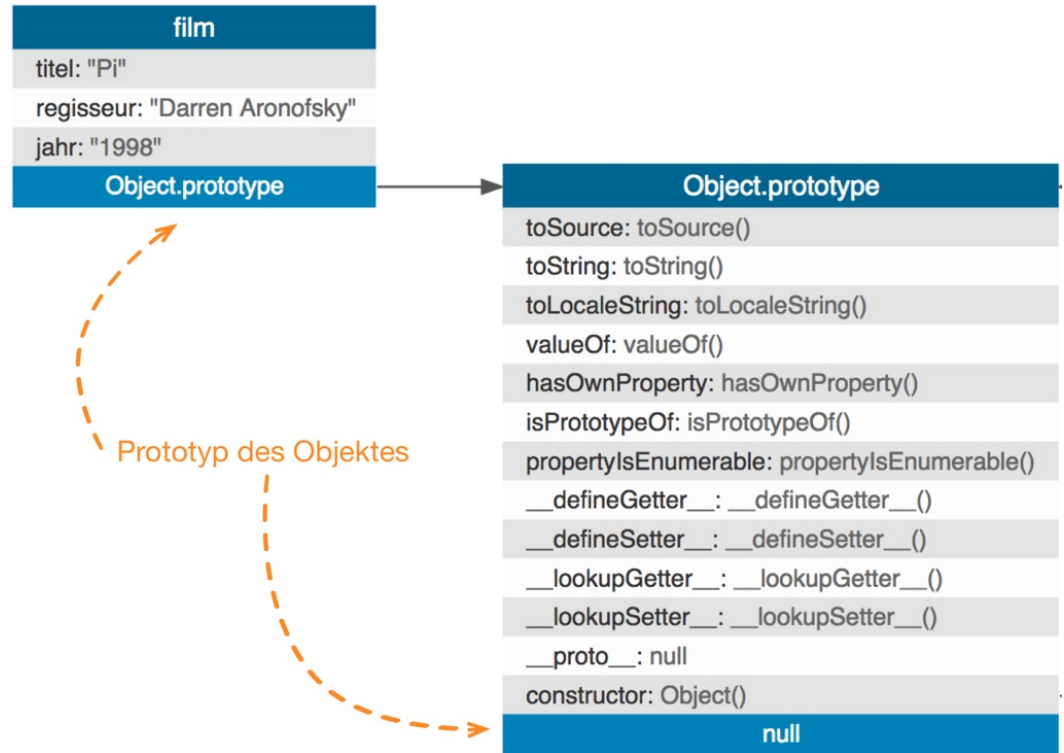
PROTOTYPENKETTE: BEISPIELE

Beispiel 1: Objekterzeugung mittels Objektliteral

JavaScript:

```
let film = {  
  titel: "Pi",  
  regisseur: "Darren Aronofsky",  
  jahr: 1998  
}
```

Prototypenkette:



Visualisierung erzeugt mit [Object Playground](#)

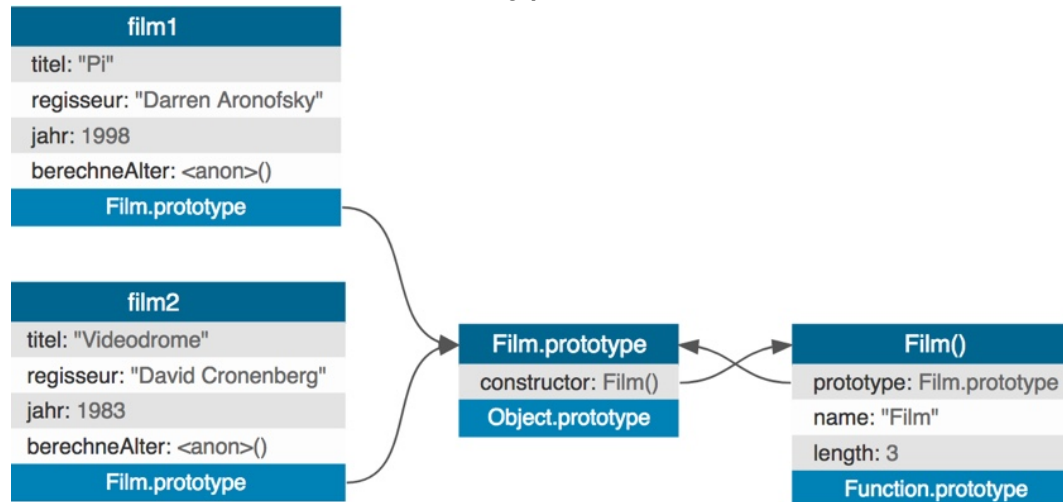
PROTOTYPENKETTE: BEISPIELE

Beispiel 2: Objekterzeugung mittels Konstruktorfunktion

JavaScript:

```
function Film(titel, regisseur, jahr) {  
  this.titel = titel;  
  this.regisseur = regisseur;  
  this.jahr = jahr;  
  this.berechneAlter = function() {  
    return new Date().getFullYear()  
      - this.jahr;  
  };  
}  
  
let film1 = new Film("Pi", "Darren Aronofsky", 1998);  
let film2 = new Film("Videodrome", "David Cronenberg", 1983);
```

Prototypenkette:



Visualisierung erzeugt mit [Object Playground](#)

PROTOTYPEN: ZUGRIFF AUF EIGENSCHAFTEN

Beim Zugriff auf eine Eigenschaft eines Objektes geht JavaScript dann wie folgt vor:

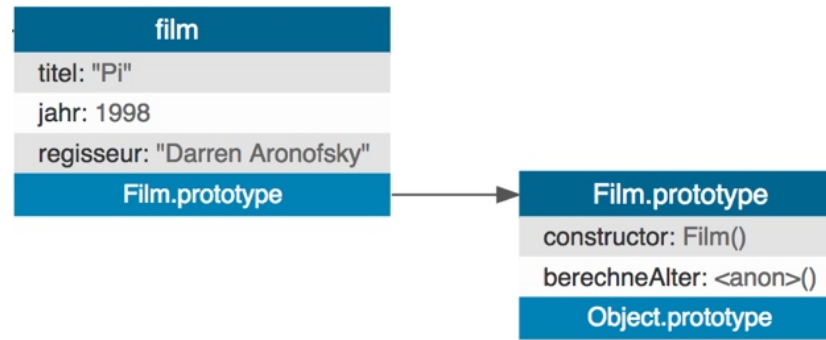
1. Zuerst wird nachgeschaut, ob das Objekt selbst die Eigenschaft besitzt (*own property*)
2. Falls nicht, wird im Prototypen des Objektes nachgeschaut
3. Falls die Eigenschaft auch dort nicht existiert, wird im Prototypen des Prototypen nachgeschaut usw.
4. Dies geschieht ggf. so lange, bis das Ende der Prototypenkette erreicht ist

ZUGRIFF AUF EIGENSCHAFTEN: BEISPIEL

JavaScript:

```
function Film(titel, regisseur, jahr) {  
  this.titel = titel;  
  this.jahr = jahr;  
  this.regisseur = regisseur;  
};  
  
/* Definition der Methode im Prototypen statt in der Konstruktorfunktion - effizienter,  
   da sich die Instanzen nun eine gemeinsame Methode teilen, statt je selbst eine separate "Instanz"  
   der Methode zu besitzen */  
Film.prototype.berechneAlter = function() {  
  return new Date().getFullYear() - this.jahr;  
};  
  
let film = new Film("Pi", "Darren Aronofsky", 1998);  
  
// Aufruf einer Methode von Film.prototype  
console.log(film.berechneAlter());  
// Aufruf einer Methode von Object.prototype  
console.log(film.toString());
```

Prototypenkette:



Visualisierung erzeugt mit [Object Playground](#)