FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Konzepte moderner Programmiersprachen (KmPS)

(Wahlfach, Modulnummer 55685)

Wintersemester 2025/2026

Prof. Dr. Andreas Claßen
(zusammen mit Prof. Dr. Heinrich Faßbender)
Fachbereich 5 Elektrotechnik und Informationstechnik
FH Aachen

JavaScript: Objekt-Orientierung

JavaScript Objekte

In JavaScript ist alles ein Objekt:

- „Literal Objects“ („normale Datenobjekte“),
- Funktionen,
- das Browser-Fenster,
- …

Es gibt „eigentlich“ keine Klassen in JavaScript: objekt-basierte Vererbung.

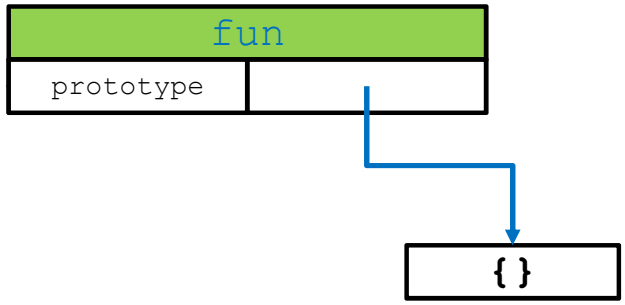
Es gibt ein „Basisobjekt“ `Object`.

`prototype` Attribut von Funktions-Objekten

Jede JavaScript Funktion hat automatisch ein „magisches Attribut“ **`prototype`**.

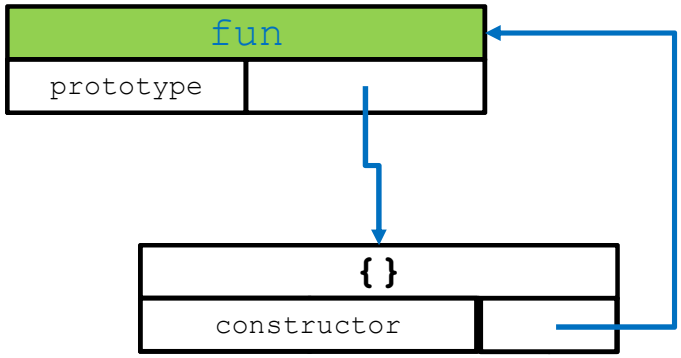
Dieses Attribut wird bei der „pseudo-klassischen“ Vererbung in JavaScript eine wichtige Rolle spielen…

Beim Erzeugen eines Funktions-Objekts wird dessen `prototype` Attribut mit einem leeren Objekt initialisiert.



`constructor` Attribut des `prototype` Objekts

Beim Anlegen einer Funktion wird in dem automatisch angelegten `prototype` Objekt ein Attribut **`constructor`** angelegt, welches auf das Funktions-Objekt zurückverweist.



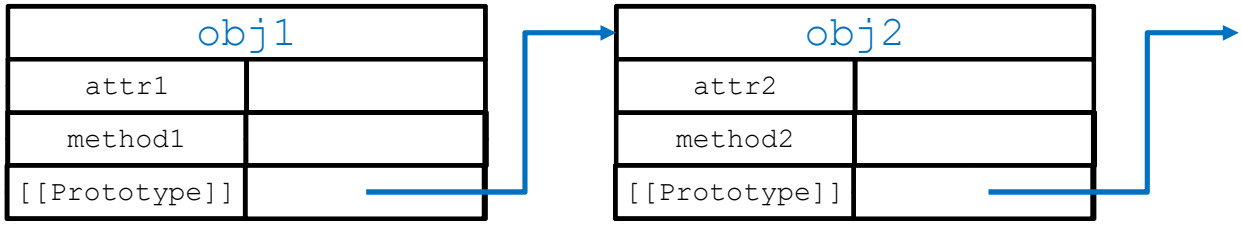
JavaScript als **Objekt-Basierte** Sprache mit **Prototypischer Vererbung**

JavaScript ist „eigentlich“ eine rein objekt-basierte Sprache, d.h. Objekte erben direkt von anderen Objekten und es gibt „eigentlich“ keine Klassen.

Diesen Ansatz bezeichnet man als **prototypische Vererbung**.

Prototypische Vererbung

Jedes Objekt kann anderes Objekt als Prototypen `[[Prototype]]` haben.

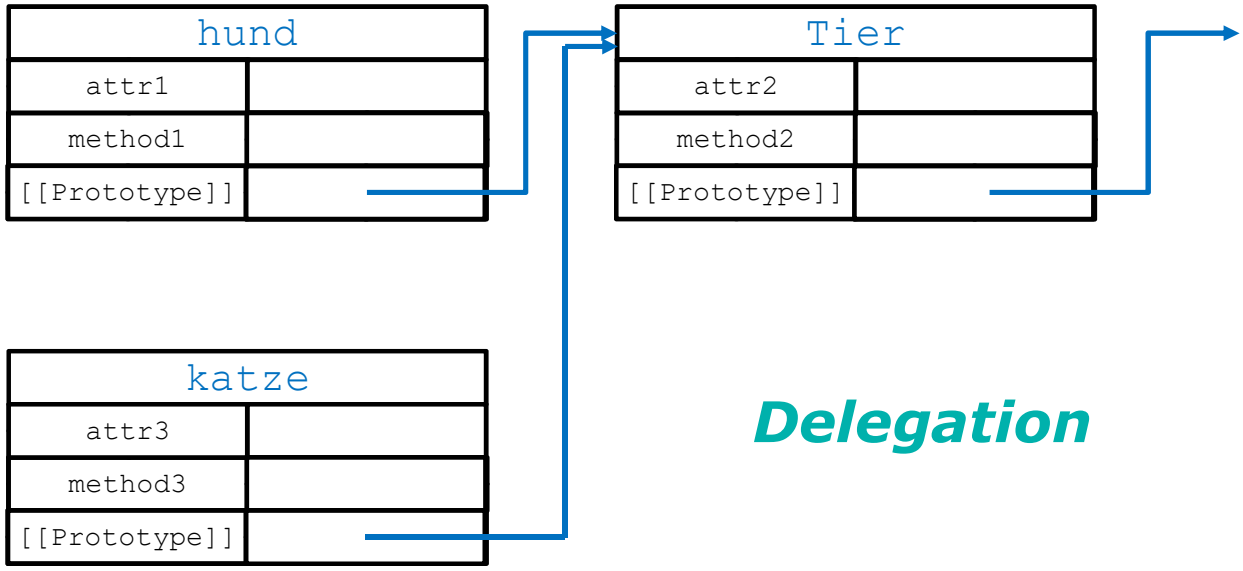


Lesen eines Attributs oder Methodenaufruf: zuerst im Objekt selbst versuchen.

Falls nicht möglich: Prototypen-Verweiskette folgen, bis Operation erfolgreich oder Kette am "Ursprungsobjekt" `Object` endet.
Erfolg in `Object` bestimmt dann, ob insgesamt erfolgreich / nicht erfolgreich.

Prototypische Vererbung

Mehrere Objekte können auf den gleichen Prototypen verweisen, der dann z.B. *zentral* Methoden für alle diese Objekte implementieren kann.

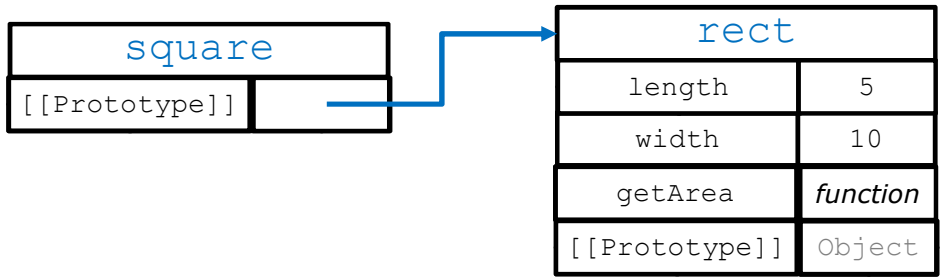


Object.create()

… erzeugt neues Objekt.
`[[Prototype]]` des neuen Objekts wird auf „Ursprungs-Objekt“ gesetzt.

Beispiel:

```
var square = Object.create(rect);
```



„Emulation“ des `Object.create()`

`Object.create()` wurde erst in ECMAScript 5 definiert!
D.h. der "Maschinerie" der prototypischen Vererbung fehlte ihre einzige, zentral benötigte Operation!
Also nicht auf einfache Art nutzbar!

Vorher konnte man aber entsprechende Funktionalität mittels einer zentralen selbst-definierten Funktion erhalten:
"Polyfill Ansatz"

```
function object_create(obj) {
    function F() {}
    F.prototype = obj;
    return new F();
}
```

Die „falschen“ Sprachkonstrukte …

Programmieren in JavaScript sollte sich „anfühlen“ wie Programmieren in Java als klassen-basierter OO Sprache.

Daher Sprachkonstrukte wie `new`.

```
var ein_buch = new Buch("Der Buchtitel");
```

Und da damit ein ausreichender Satz „OO-Konstrukte“ existierte, wurde kein Befehl mehr aufgenommen, um rein objekt-basierte Vererbung elegant umzusetzen …
D.h. kein ***`create()`*** *, um aus* ***`Object`*** *weitere Objekte zu erzeugen und per Prototyp zu verlinken!*

„OO Methodiken“ in JavaScript

Somit in JavaScript zwei „OO Methodiken“:

1. „**Pseudo-klassische Objekt-Orientierung**“ ähnlich zu Java:
   "Klassen" und "Klassen-basierte Vererbung" umsetzbar,
   unterstützt durch Sprachkonstrukte (`new`, …).
   Aber entspricht nicht der technischen Realisierung.
   Wird bei der Programmierung üblicherweise genutzt.

2. Die „**objekt-basierte Objekt-Orientierung**“:
   Entspricht der technischen Realisierung.
   Nur Objekte, keine Klassen.
   Sprachkonstrukte (`clone`, …) erst seit ECMAScript 5 (vorher „Emulation“ möglich).
   Prototypische Vererbung.
   Wird selten auf Programmierer-Ebene (d.h. im Anwendungs-Code) genutzt.

Pseudo-Klassische OO Methodik in JavaScript

- Funktions-Objekte in der Rolle von Klassen und als Konstruktoren,
- **new** zum Erzeugen von Instanz-Objekten.

```
function Buch(titel) { … }

var ein_buch = new Buch("Der Buchtitel");
```

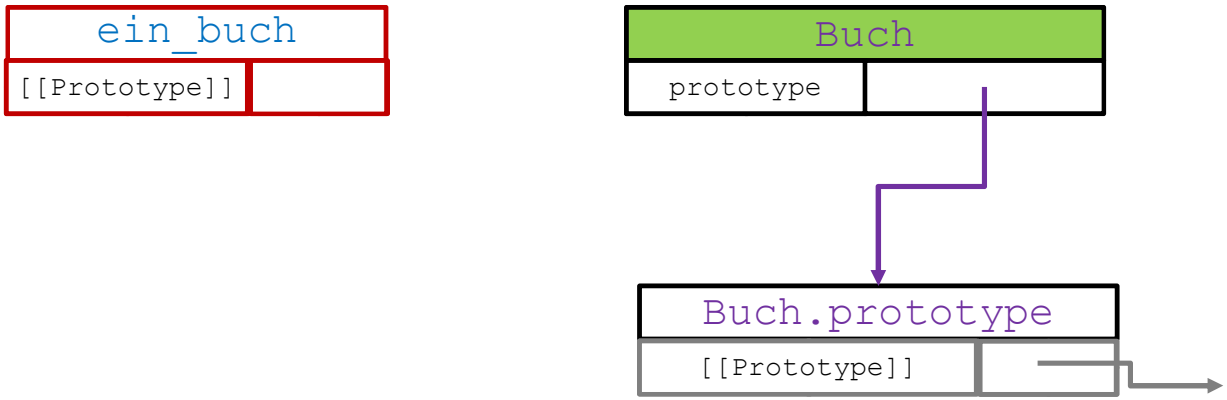
Somit ähnlich zu Java, was Erzeugung von Objekten betrifft.

Was der `new` Befehl leistet …

Am Beispiel `var ein_buch = new Buch("…");`

1. `new` legt ein neues Objekt an.

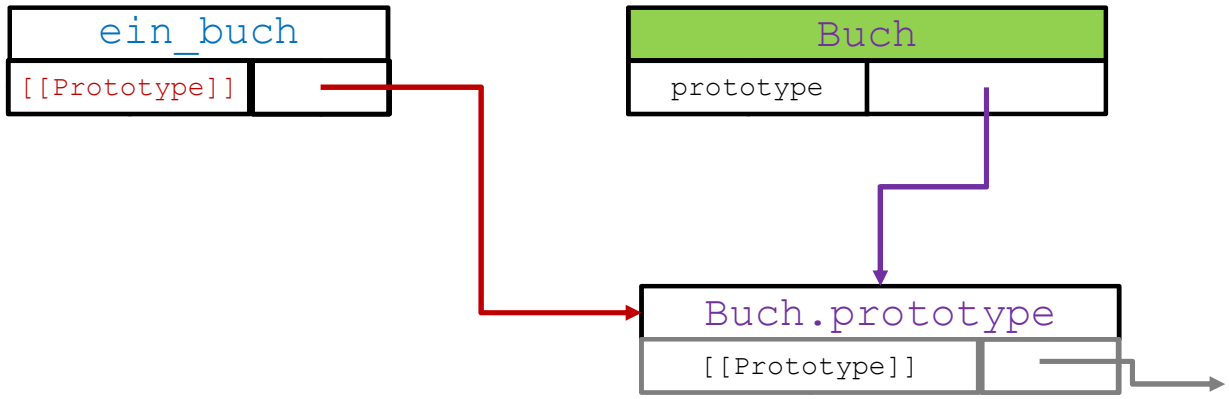
```
function Buch(titel) { … }

var ein_buch = new Buch("…");
```



Was der `new` Befehl leistet …

Am Beispiel `new Buch("…");`

2. `new` ändert `[[Prototype]]` des neuen Objekts auf das Objekt im `prototype` Attribut der Konstruktor-Funktion.

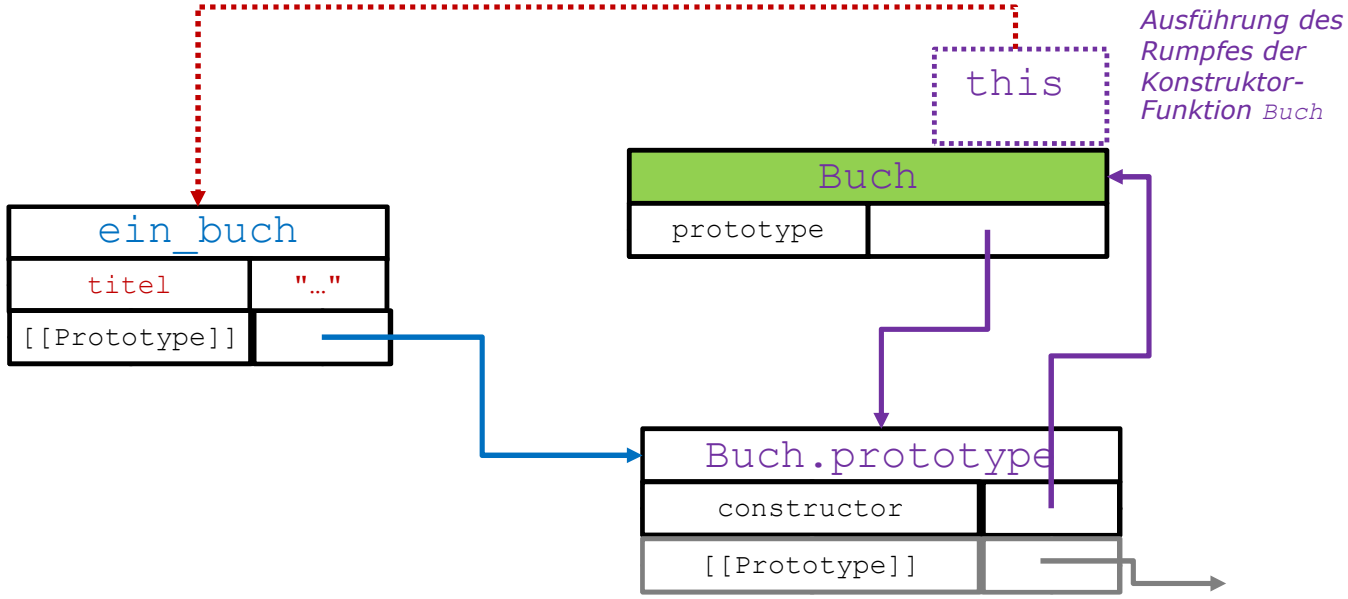


Was der `new` Befehl leistet …

Am Beispiel `var ein_buch = new Buch("…");`

3. Die Konstruktor-Funktion wird aufgerufen, neues Objekt als Wert von `this`.

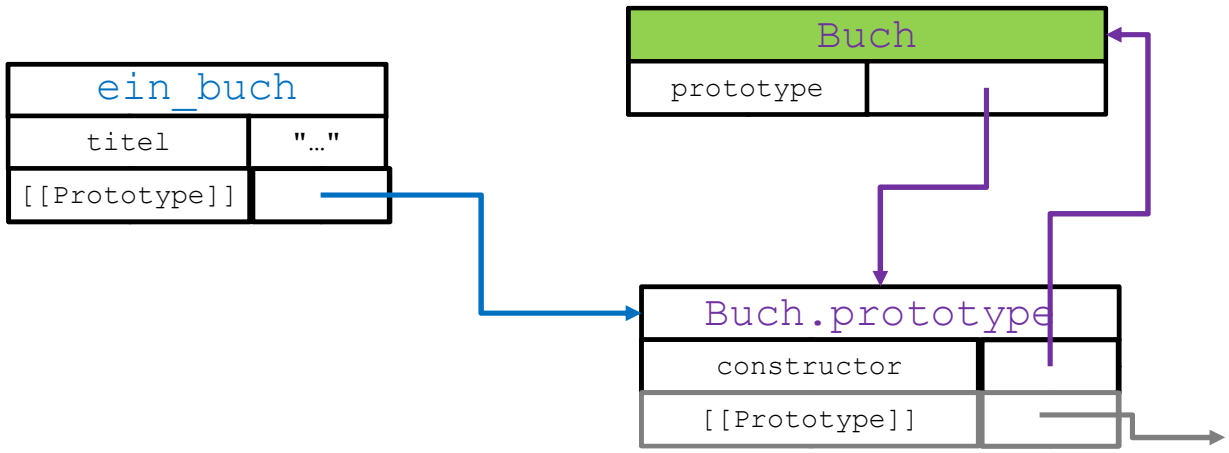
```
function Buch(titel) {
    this.titel = titel;
}
```



Was der `new` Befehl leistet …

Am Beispiel `new Buch("…");`

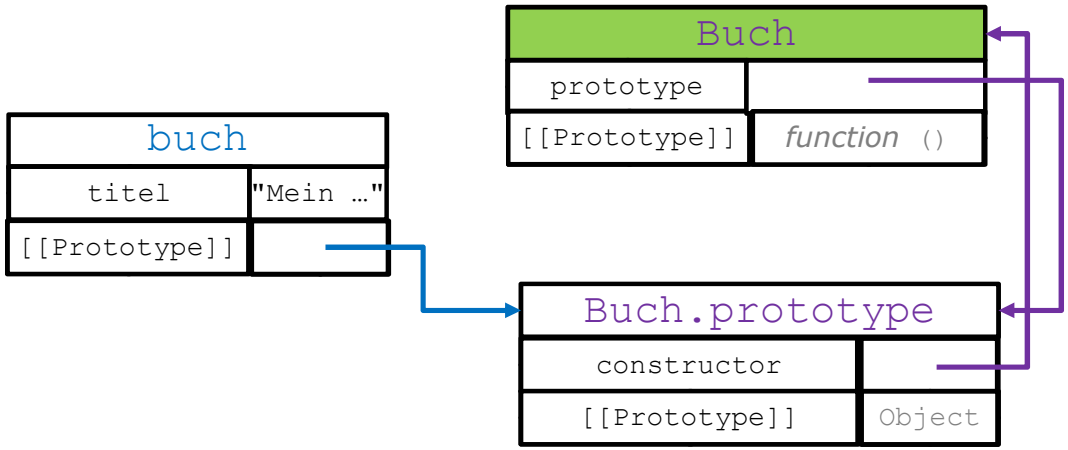
4. Hat Konstruktor-Funktion keinen Rückgabewert, so wird neues Objekt zum Resultatwert.



Was der `new` Befehl leistet …

Beispiel:

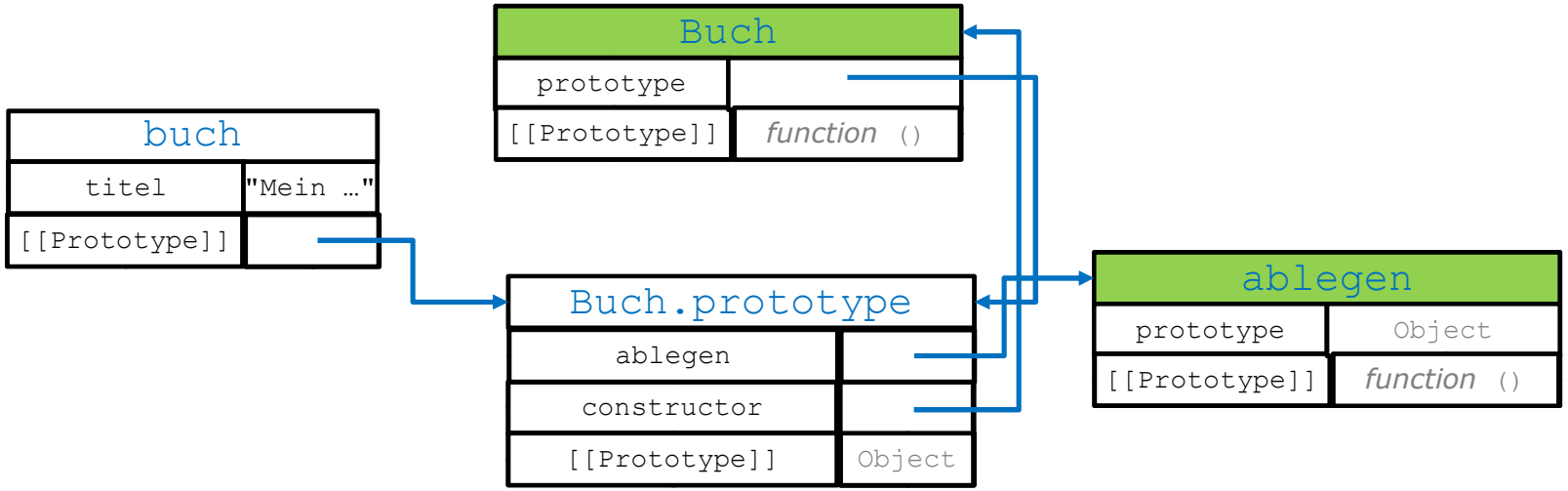
```
function Buch(titel) {
    this.titel = titel;
}

var buch = new Buch("Mein Buch");
```



Pseudo-Klassische Methodik: `prototype` Objekt der Konstruktor-Fkt als "Methodensammler"

Beispiel:

```
function Buch (titel) {
    this.titel = titel;
}
Buch.prototype.ablegen = function() {
    console.log("Im Regal ablegen.");
}
var buch = new Buch("Mein Buch");
buch.ablegen();
```



Pseudo-Klassische ***Vererbung***

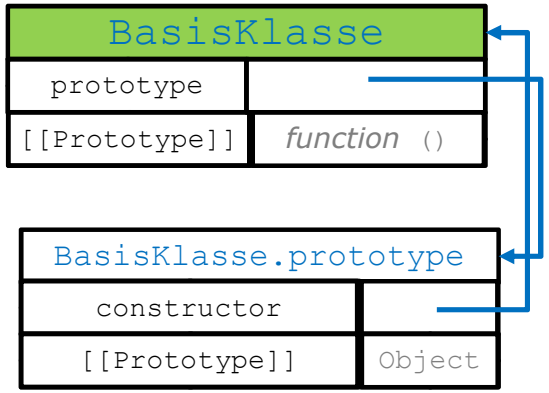
… muss durch "komisches Programmier-Pattern" erzielt werden.

Beispiel:

```
function BasisKlasse() {};

function AbgeleiteteKlasse() {
    // ggf. BasisKlasse.call(this);
};

AbgeleiteteKlasse.prototype = new BasisKlasse();
AbgeleiteteKlasse.prototype.constructor = AbgeleiteteKlasse;
```

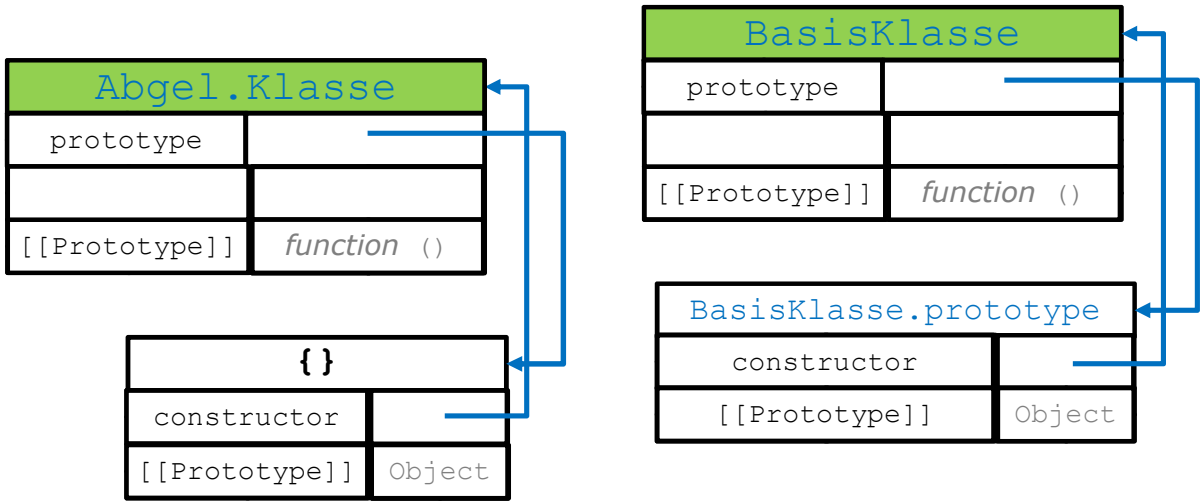
Pseudo-Klassische Vererbung

```
function BasisKlasse() {};
```



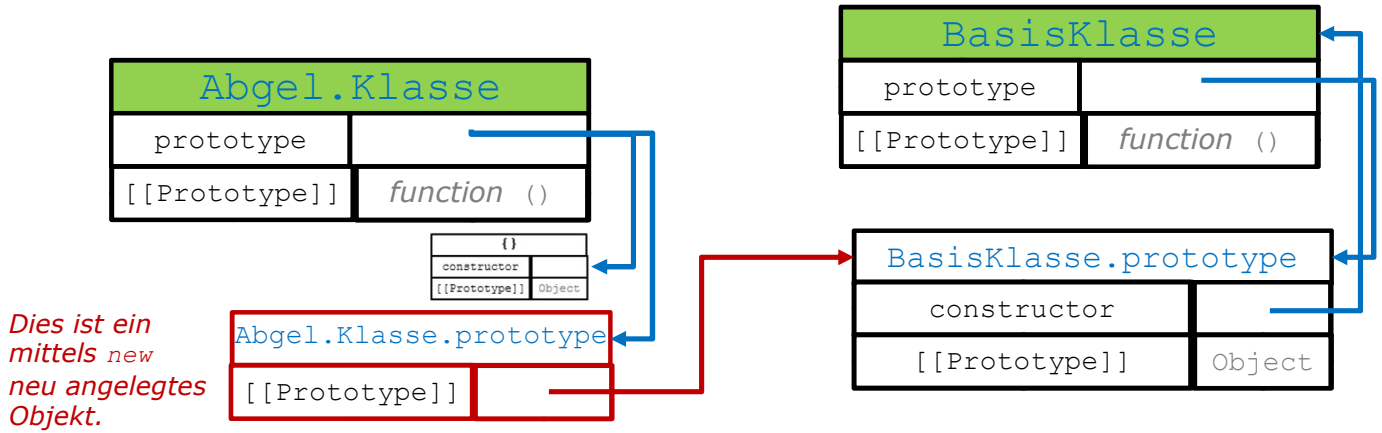
Pseudo-Klassische Vererbung

```
function BasisKlasse() {};

function AbgeleiteteKlasse() {
    // ggf. BasisKlasse.call(this);
};
```



Pseudo-Klassische Vererbung

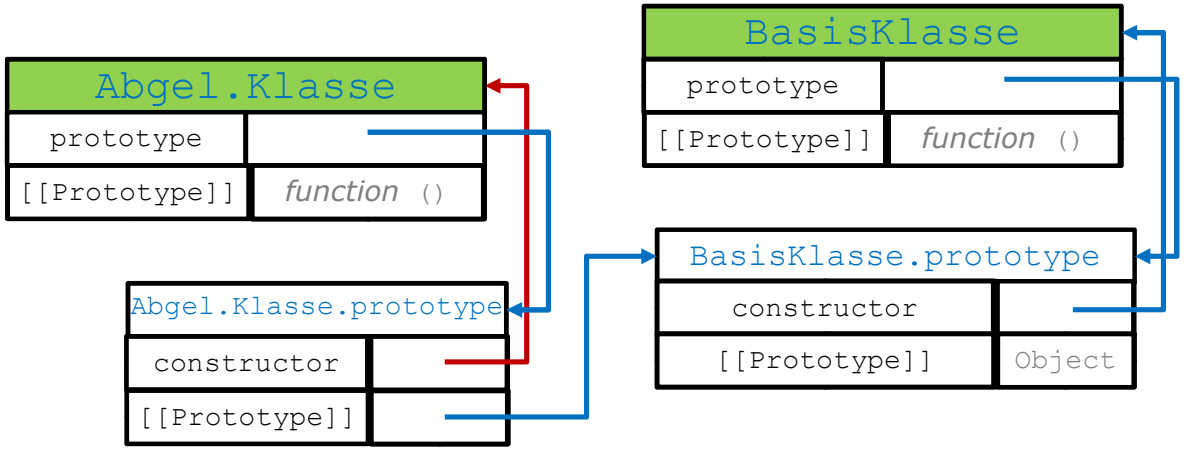
```
AbgeleiteteKlasse.prototype = new BasisKlasse();
```



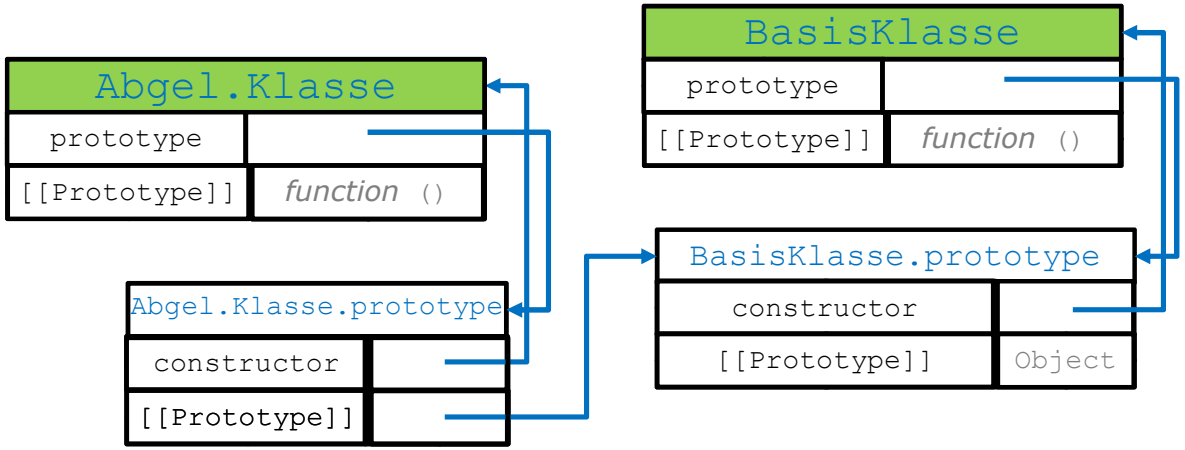
Bisheriges `prototype` Objekt von `AbgeleiteteKlasse` wird garbage-collected

Pseudo-Klassische Vererbung

```
AbgeleiteteKlasse.prototype.constructor = AbgeleiteteKlasse;
```



Pseudo-Klassische Vererbung: "Resultat"



Weitere Pseudo-Klassische Patterns

Z.B.:

- David Flanagan, Autor des sehr bekannten JavaScript Buches *JavaScript: The Definitive Guide* („TDG“, O'Reilly):
  `Class()` Funktion, welche die wesentlichen Aspekte der klassen-basierten Vererbung realisiert.

- John Resig, Autor der sehr bekannten *jQuery* JavaScript Library: `Class` Object mit Methoden für klassen-basierte Vererbung, insbesondere `extend()`.

- Dean Edwards, Autor der *Base.js* JavaScript Library: `Base` Object mit Methoden für klassen-basierte Vererbung, insbesondere `extend()`.

- …

Pseudo-Klassische Patterns: Flanagan

```
var A = Class({
    name: "A",
    init: function( p0, p1 ) {
        this.a = p0;
        this.b = p1;
    },
    methods: {
        toString: function() {
            return 'A{' +
                'a=' + this.a +
                ',b=' + this.b +
            '}';
        }
    }
});
```

```
var B = Class({
    name: "B",
    extend: A,
    init: function( p0, p1, p2, p3 ) {
        this.c = p2;
        this.d = p3;
    },
    methods: {
        toString: function() {
            return 'B{' +
            chain(this,arguments) +
            ',c=' + this.c +
            ',d=' + this.d + ")";
        },
    }
});
```

Pseudo-Klassische Patterns: Resig

```
var A = Class.extend(
    {
        init: function ( p1, p2 ) {
            this.a = p1;
            this.b = p2;
        },
        toString: function () {
            return 'A{' +
                'a=' + this.a +
                ',b=' + this.b +
            '}';
        }
    }
);
```

```
var B = A.extend(
    {
        init: function ( p1, p2, p3, p4 ) {
            this._super( p1, p2 );
            this.c = p3;
            this.d = p4;
        },
        toString: function () {
            return 'B{' +
                this._super() +
                ',c=' + this.c +
                ',d=' + this.d +
            '}';
        }
    }
);
```

Klassen und Vererbung seit ECMAScript 6

Seit ECMAScript 6: Klassen, Properties, Vererbung direkt programmierbar.

Schlüsselwort `class`.

Methoden innerhalb der Klassendefinition.

Konstruktor mittels `constructor`.

Ableiten von einer anderen Klasse mittels `extends`.

`get` und `set` für Properties .

Klassen in ECMAScript 6 (ECMAScript 2015)

"JavaScript Klassen … sind syntaktischer Zucker für das bestehende, auf Prototypen basierende, Vererbungsmodell …

… führt kein neues OOP-Modell in die Sprache ein."

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen