WBE: JAVASCRIPT

PROTOTYPEN VON OBJEKTEN

ÜBERSICHT

- Mehr zu JavaScript-Objekten
- Prototypen und this
- Konstruktoren und Vererbung
- Gewohntere Syntax: Klassen
- Test-Driven Development

ÜBERSICHT

- Mehr zu JavaScript-Objekten
- Prototypen und this
- Konstruktoren und Vererbung
- Gewohntere Syntax: Klassen
- Test-Driven Development

WERTE-DATENTYPEN

- Zahlen, Strings und Wahrheitswerte sind *Wertetypen*
- Sie sind unveränderlich
- Zuweisung kann wie Kopieren behandelt werden

```
1 let msg = "Hello developers!"
2 let greeting = msg
3 greeting += "!!"
4
5 console.log(greeting)     /* → 'Hello developers!!!' */
6 console.log(msg)          /* → 'Hello developers!'   */
```

Speaker notes

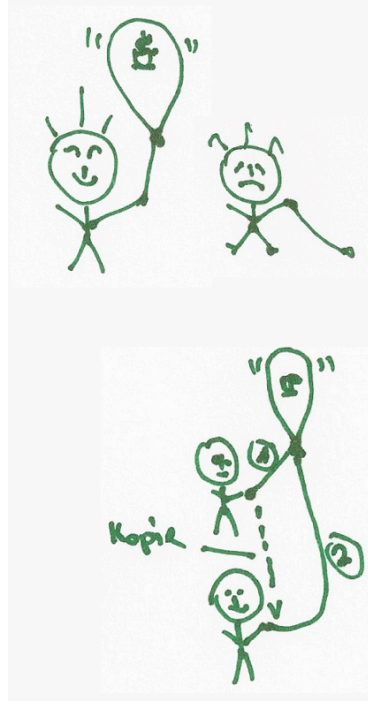
String-Anpassungen erfolgen also nicht an Ort und Stelle. Es wird ein neuer String erzeugt und dieser muss wieder zugewiesen werden. Nach

```
let msg = "Hello developers!"
```

kann man also sicher sein, dass sich der Inhalt von `msg` nicht ändert, solange der Variablen nichts neues zugewiesen wird. Das gilt auch für die Parameterübergabe: da ein übergebener String unveränderbar ist, kann angenommen werden, dass die Funktion mit einer Kopie der Zeichenkette arbeitet.

REFERENZ-DATENTYPEN

- Objekte und Arrays sind *Referenz-Datentypen*
- Sie sind jederzeit veränderbar
- Es werden Referenzen zugewiesen




```
1 let obj = { message: "loading..." }
2 let anotherObj = obj
3 anotherObj.message = "ready"
4
5 console.log(obj)      /* → { message: 'ready' } */
```

REFERENZ-DATENTYPEN

- Objekt- und Array-Literale legen neue Objekte an
- `==` und `===` vergleichen die Referenzen
- `const` heisst: Referenz kann nicht geändert werden

```
> const a = [1, 2, 3]
> const b = [1, 2, 3]
> const c = a

> a == b
false
> a == c
true
> c[0] = 99
> a
[ 99, 2, 3 ]
```

```
> const obj = { message: "loading..." }

> obj.message = "ready"
'ready'

> obj = {}
Uncaught TypeError: Assignment to
constant variable.
```

Speaker notes

Mit `const` kann die Referenz als Konstante definiert werden. Das Ziel ist dadurch immer noch veränderbar, nicht aber die Referenz selber.

Mit `Object.freeze` können die Attribute eines Objekts unveränderlich gemacht werden. Dies gilt für die oberste Ebene der Attribute. Wenn man eine ganze Objekthierarchie unveränderbar machen möchte, kann man sich leicht ein `deepFreeze` o.ä. schreiben.

```
> const obj = { prop:42, nested: {} }
> Object.freeze(obj)
> obj.prop = 33
> obj
{ prop: 42, nested: {} }

> obj.nested.attr = 1
> obj
{ prop: 42, nested: { attr: 1 } }

> ((obj)=> {"use strict"; obj.prop = 33;})(obj)
Uncaught TypeError: Cannot assign to read only property 'prop' of object '#<Obje
    at REPL14:1:34
```

WERTE- UND REFERENZTYPEN

- Objekte sind also Referenztypen
- Das gilt auch für Arrays und Funktionen
- Referenzen vs. Werte vergleichen:

```
> [ []==[], {}=={}, (()=>{})==(()=>{}) ]
[ false, false, false ]

> [ 3.5==3.5, "abc"=="abc", false==false ]
[ true, true, true ]
```

Speaker notes

Konsequenz: Strings können in JavaScript jederzeit mit == oder === verglichen und generell wie Werte behandelt werden.

Bei der Übergabe von *Objekten* an eine Funktion muss damit gerechnet werden, dass diese in der Funktion geändert werden. Übrigens: Sowohl Funktionen als auch Arrays erben von `Object.prototype`. Mehr dazu später.

In vielen Situationen möchte man auch Objekte als Werte betrachten können. Das setzt voraus, dass man Objekte nicht verändert bzw. bei Änderungen immer neue Objekte erzeugt. Das hat zahlreiche Vorteile, geht jedoch auf Kosten von Speicherverbrauch und Rechenzeit. Unveränderbare Datenstrukturen sind das Ziel der Bibliothek immutable.js von FaceBook:

https://immutable-js.github.io/immutable-js/

In ECMAScript 2023 wurden einige Array-Methoden hinzugefügt, welche das ursprüngliche Array nicht verändern und ein neues Array liefern:

- `toSorted` liefert ein neues, sortiertes Array
- `toReversed` kehrt ein Array um und liefert das Ergebnis als neues Array
- `toSpliced` liefert ein neues Array mit geänderter Teilsequenz
- `with` liefert ein neues Array mit einer überschriebenen Position

```
const languages = ["JavaScript", "TypeScript", "CoffeeScript"]
const updated = languages.with(2, "WebAssembly")
```

https://www.sonarsource.com/blog/es2023-new-array-copying-methods-javascript/

ATTRIBUTE

- Wie Objekte und Arrays können auch Werte in JavaScript Attribute (bzw. Methoden) haben
- Ausnahmen: `null`, `undefined`

```
> "Zeichenkette".length
12
> "Zeichenkette"["length"]
12
> "Zeichenkette".toUpperCase()
'ZEICHENKETTE'
```

```
> 1.5.toString()
'1.5'
> (1/3).toPrecision(4)
'0.3333'
```

Speaker notes

Methoden werden wohl selten auf Zahlenliteralen aufgerufen wie im Beispiel. Wenn das trotzdem geschieht, muss man darauf achten, dass es nicht zu Verwechslung mit dem Dezimalpunkt kommt:

```
> 1.5.toString()
'1.5'
> 1.toString()
Uncaught SyntaxError: Invalid or unexpected token
> 1..toString()
'1'
> (1).toString()
'1'
```

ATTRIBUTE

- Attribute von Wertetypen sind unveränderlich
- Zuweisung neuer Attribute zu Wertetypen nicht möglich
- Objekten (auch Arrays, Funktionen) können aber jederzeit Attribute zugewiesen werden

```
> const square = n => n*n

> square.doc = "Quadratfunktion"
'Quadratfunktion'

> square(3)
9

> square.doc
'Quadratfunktion'
```

Speaker notes

ARRAYS UND ATTRIBUTE

- Die (normalen) Attribute eines Arrays sind ganze Zahlen >=0
- Wird etwas anderes als Index angegeben, wird ein Attribut hinzugefügt

```
> let arr = [1, 2, 3]
> arr[-1] = 4
> arr['key'] = 'value'

> arr
[ 1, 2, 3 ]
> arr.key
'value'
```

Je nach Implementierung werden die Attribute '-1' und 'key' mit dem Array ausgegeben. Unter bestimmten Node.js-Versionen zum Beispiel:

```
> arr
[ 1, 2, 3, '-1': 4, key: 'value' ]
```

Tatsächlich kann man alle Attribute, numerische Array-Attribute und andere Attribute, in ein *normales* Objekt überführen:

```
> Object.assign({}, arr)
{ '0': 1, '1': 2, '2': 3, '-1': 4, key: 'value' }
```

Auch lässt sich der reine Array-Inhalt extrahieren:

```
> Array.from(arr)
[1, 2, 3]
```

WERTETYPEN UND ATTRIBUTE

Die Zuweisung neuer Attribute zu Wertetypen ist nicht möglich. Im Strict Mode gibt's eine TypeError-Exception, ansonsten wird die Zuweisung ignoriert. Beispiel:

```
> let name = "John Johnson"
undefined
> name.doc = "Das ist sein Name"
'Das ist sein Name'
> name.doc
undefined
```

VORDEFINIERTE OBJEKTE

- `String`, `Number`, `Boolean`: Wrapper um Basistypen
- `Math`: mathematische Funktionen und Konstanten
- `Date`: Datums- und Zeitangaben
- `Map`, `Set`: Schlüssel/Wert Zuordnung bzw. Menge
- `RegExp`: reguläre Ausdrücke
- `Object`: allgemein Objekte

Zahlreiche weitere vordefinierte Objekte...

MDN: Standard built-in objects

Kleine Auswahl auf den folgenden Slides...

String

- Strings sind in JavaScript ein primitiver Datentyp
- Erzeugt durch String-Literale `"..."`, `'...'`, `` `...` ``
- String-Methoden sind in `String.prototype` definiert (mehr zu Prototypen später)

```
> String.prototype.slice
[Function: slice]

> "Hello World".slice(0, 5)
'Hello'
```

Speaker notes

Tatsächlich gibt es auch String-Objekte, da String auch ein Konstruktor ist. In den meisten Fällen können primitive Strings und String-Objekte gleich verwendet werden. String-Objekte sind allerdings Referenztypen und können um Attribute ergänzt werden. Mit der `valueOf`-Methode kann aus einem String-Objekt ein primitiver String gewonnen werden.

```
> let sPrim = 'foo'
> let sObj = new String(sPrim)

> console.log(typeof sPrim)
"string"
> console.log(typeof sObj)
"object"

> sObj
[String: 'foo']
> sObj.valueOf()
'foo'
```

Man könnte sich fragen, warum `String.prototype.slice` eine Funktion liefert, `String.prototype` jedoch ein leeres Objekt. Das liegt daran, dass nicht alle Properties von Objekten *enumerable* sind. Sie können jedoch durch `Object.getOwnPropertyNames` in ein Array ausgegeben werden:

```
> String.prototype.slice
[Function: slice]
> String.prototype
{}
> Object.getOwnPropertyNames(String.prototype)
[
  'length',      'constructor',       'anchor',
  'big',         'blink',             'bold',
...
]
```

STRING-METHODEN

- `slice`: Ausschnitt aus einem String (vgl. Arrays)
- `indexOf`: Position eines Substrings (vgl. Arrays)
- `trim`: Whitespace am Anfang und Ende entfernen
- `padStart`: vorne Auffüllen bis zu bestimmter Länge
- `split`: Auftrennen, liefert Array von Strings
- `join`: Array von Strings zusammenfügen
- `repeat`: String mehrfach wiederholen

MDN: String

Speaker notes

Auf einzelne Zeichen eines Strings kann auch über den Index zugegriffen werden:

```
> let msg = 'Hello'
> msg[0]
'H'
> Object.keys(msg)
[ '0', '1', '2', '3', '4' ]
```

Seit ECMAScript 2022 ist ein Zugriff auch mit der `at`-Methode möglich. >Im Gegensatz zur []-Notation sind auch negative Werte möglich:

```
> let msg = 'Hello'
> msg.at(0)
'H'
> msg.at(-1)
'o'
```

Number

- Methoden und Konstanten von `Number`
- Methoden von `Number.prototype` können auf einzelne Zahlen angewendet werden

```
> Number.MAX_VALUE
1.7976931348623157e+308

> Number.isInteger(1.5)
false

> 3500.75.toLocaleString('de-DE')
'3.500,75'
```

MDN: Number

Speaker notes

Für `Number` gilt ebenso wie für `String`: es kann auch als Konstruktor verwendet werden, um Objekte zu erstellen, mit `valueOf` kann man den primitiven Typ erhalten. Das wird man aber kaum je brauchen. `Number` kann auch verwendet werden, um zu Zahlen zu konvertieren:

```
let n = Number("12")
```

Math

- Methoden und Konstanten als Attribute des `Math`-Objekts
- Objekt als Namensraum für Methoden und Konstanten
- Gleich wie `String` und `Number` ist `Math` built-in

```
> Math.E
2.718281828459045

> Math.exp(1)
2.718281828459045

> Math.min(5, 2, 7, -4, 12)
-4
```

```
> Math.sin(0.5)
0.479425538604203

> Math.round(3.7)
4

> Math.random()
0.04802545627381716
```

MDN: Math

Speaker notes

Im Gegensatz zu `Number` und `String` kann `Math` nicht als Konstruktor verwendet werden. `Math` ist ein vordefiniertes Objekt mit einigen Attributen und Methoden.

Um von einer Zahl mit Nachkommastellen zu einer Ganzzahl zu kommen, gibt es verschiedene Methoden von `Math`:

- `Math.floor`: nächst kleinere Ganzzahl
- `Math.ceiling`: nächst grössere Ganzzahl
- `Math.round`: Runden zur nächsten Ganzzahl

Weitere Methoden:

- `Math.abs`: positiver Wert
- `Math.log`: natürlicher Logarithmus
- `Math.max`: grösstes der Argumente
- `Math.min`: kleinstes der Argumente
- `Math.pow`: Zahl hoch Exponent
- `Math.sin`: Sinus
- `Math.sqrt`: Quadratwurzel

Weitere Konstanten:

- `LN2`: natürlicher Logarithmus von 2
- `LN10`: natürlicher Logarithmus von 10
- `LOG2E`: Logarithmus von 2
- `LOG10E`: Logarithmus von 10
- `PI`: Konstante PI
- `SQRT1_2`: Quadratwurzel aus 0.5
- `SQRT2`: Quadratwurzel aus 2

Date

```
let now = new Date()

console.log(now)                    /* → 2021-10-09T15:43:21.753Z */
console.log(now.getHours())         /* → 17                       */
console.log(now.getUTCHours())      /* → 15                       */
console.log(now.getTime())          /* → 1633794201753            */

now.setFullYear(now.getFullYear()+1)
console.log(now.toString())
// 'Sun Oct 09 2022 17:43:21 GMT+0200 (Mitteleuropäische Sommerzeit)'
```

MDN: Date

Speaker notes

Es gibt ein paar Merkwürdigkeiten des JavaScript `Date`-Konstruktors. Zum Beispiel liefert `getYear()` die Anzahl Jahre seit 1900, für 1995 also 95. Erstaunlich, dass man nicht daran gedacht hat, dass das Jahr 2000 kommen wird und das Jahr dann 100 liefert. Einige Webseiten haben dann auch am 1.1.2000 das Datum als 1.1.19100 angegeben. Schliesslich war der Fix dann, die Methode `getFullYear()` einzuführen.

Ein zweiter Fallstrick ist der Monat. Hier wird mit 0 begonnen zu zählen, der Januar ist also Monat 0, Dezember ist Monat 11.

Da `Date` immer wieder Probleme macht, gibt es einen Vorschlag, ECMAScript mit `Temporal` eine neue API hinzuzufügen. Aus dem Vorschlag:

„Date has been a long-standing pain point in ECMAScript. This is a proposal for Temporal, a global Object that acts as a top-level namespace (like Math), that brings a modern date/time API to the ECMAScript language.”
https://tc39.es/proposal-temporal/docs/index.html

ÜBERSICHT

- Mehr zu JavaScript-Objekten
- Prototypen und this
- Konstruktoren und Vererbung
- Gewohntere Syntax: Klassen
- Test-Driven Development

METHODEN

```
1 let player = {
2   sayHello: function () { return  "Hello" },  /* ausführlich    */
3   sayHi() { return  "Hi" },                   /* abgekürzt      */
4   sayBye: () => "Bye",                        /* Pfeilnotation  */
5 }
```

- Verschiedene Notationen für Methoden
- Abgekürzte Variante seit ECMAScript 2015 möglich

this

- Bezieht sich auf das aktuelle Objekt
- Was das heisst, ist nicht immer ganz klar
- Bedeutung ist abhängig davon, wo es vorkommt
  - Methodenaufruf (method invocation)
  - Funktionsaufruf (function invocation)
  - Mit `apply`, `call` oder `bind` festgelegt
  - Konstruktoraufruf

THIS: METHODENAUFRUF

```
1 function speak (line) {
2   console.log(`The ${this.type} rabbit says '${line}'`)
3 }
4 let whiteRabbit = {type: "white", speak}
5 let hungryRabbit = {type: "hungry", speak}
6
7 hungryRabbit.speak("I could use a carrot right now.")
8 // → The hungry rabbit says 'I could use a carrot right now.'
```

- `this` in einer Funktion ist abhängig von Art des Aufrufs
- Aufruf als Methode eines Objekts: `this` ist das Objekt

THIS: FUNKTIONSAUFRUF

```
1 function speak (line) {
2   console.log(`The ${this.type} rabbit says '${line}'`)
3 }
4
5 speak("I could use a carrot right now.")
6 // → The undefined rabbit says 'I could use a carrot right now.'
```

- Hier ist `this` das globale Objekt (Node REPL: `global`)
- Es hat kein `type`-Attribut, daher wird `undefined` eingesetzt
- Dies ist praktisch immer ein Programmierfehler

Speaker notes

Was das globale Objekt ist, ist abhängig von der Laufzeitumgebung. Im Browser ist das globale Objekt `window`. Es repräsentiert das aktuell geöffnete Browserfenster (bzw. Tab). In Node.js heisst das globale Objekt `global`.

In einem CommonJS-Modul ist `this` auf oberster Ebene an `module.exports` gebunden Das können Sie verifizieren, indem Sie die folgende Zeilen in einem CommonJS-Modul einmal auf oberster Ebene und einmal in einer Funktion ablegen:

```
console.log(this == module.exports)
console.log(this === global)
```

Im Modul ist die erste Ausgabe `true` und die zweite `false`. In der Funktion ist es umgekehrt. Konsequenz: Werte (Funktionen...) können in einem CommonJS-Modul auf mehrere Arten exportiert werden:

```
module.exports.value = value
exports.value = value
this.value = value
```

Unabhängig von der aktuellen Laufzeitumgebung kann auf das globale Objekt in der Regel mit `globalThis` zugegriffen werden:

Note: `globalThis` is generally the same concept as the global object (i.e. adding properties to `globalThis` makes them global variables) — this is the case for browsers and Node — but hosts are allowed to provide a different value for `globalThis` that's unrelated to the global object.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis

STRICT MODE

- Behebt einige potenzielle Fehlerquellen in JavaScript
- Aktiviert am Anfang des Scripts / der Funktion durch `"use strict"`
- Im strict mode ist `this` bei Funktionsaufruf `undefined`

```
1 "use strict"
2
3 function speak (line) {
4   console.log(`The ${this.type} rabbit says '${line}'`)
5 }
6
7 speak("I could use a carrot right now.")
8 // → TypeError: Cannot read property 'type' of undefined
```

Speaker notes

Für den strict-Modus ein neues Kommando einzuführen, wäre nicht gegangen, da ältere Browser damit Probleme hätten. Daher hat man sich für das Einfügen des Strings "use strict" entschieden. Ältere Browser ignorieren diesen String einfach, da er nicht weiter verwendet wird.

Im nicht strikten Modus würde sich `this` hier wieder auf das globale Objekt beziehen und wenn dieses kein `type`-Attribut hat einfach `undefined` in den String einfügen. Im strikten Modus hat aber bereits `this` keinen Wert bzw. ist `undefined`, so dass `this.type` eine Exception auslöst.

Im Beispiel erfolgt nur ein lesender Zugriff über `this`. Bei schreibendem Zugriff würden hier im nicht strikten Modus neue Attribute im globalen Objekt angelegt, was man ziemlich sicher nicht beabsichtigt hat.

Fehler werden somit im strikten Modus leichter gefunden.

Der strict-Modus kann auch auf eine Funktion beschränkt werden. Dies würde hier genauso funktionieren:

```
function speak (line) {
  "use strict"
  console.log(`The ${this.type} rabbit says '${line}'`)
}

speak("I could use a carrot right now.")
// → TypeError: Cannot read property 'type' of undefined
```

call, apply

- Methoden `call` und `apply` von Funktionen
- Erstes Argument: Wert von `this` in der Funktion
- Weitere Argumente von `call`: Argumente der Funktion
- Weiteres Argument von `apply`: Array mit den Argumenten

```
1 function speak (line) {
2   console.log(`The ${this.type} rabbit says '${line}'`)
3 }
4 let hungryRabbit = {type: "hungry"}
5
6 speak.call(hungryRabbit, "Burp!")
7 // → The hungry rabbit says 'Burp!'
```

Speaker notes

Oder mit `apply`:

```
speak.apply(hungryRabbit, ["Burp!"]);
// → The hungry rabbit says 'Burp!'
```

Funktionen haben also Methoden – etwas gewöhnungsbedürftig.

Übrigens: `speak` muss in diesem Fall nicht Methode des Objekts sein.

bind

- Noch eine Methode von Funktionen: `bind`
- Erzeugt neue Funktion mit gebundenem `this`
- Auch weitere Argumente können gebunden werden

```
function speak (line) {
  console.log(`The ${this.type} rabbit says '${line}'`)
}
let hungryRabbit = {type: "hungry"}

let boundSpeak = speak.bind(hungryRabbit)
boundSpeak("Burp!")
// → The hungry rabbit says 'Burp!'
```

FUNKTIONEN IN PFEILNOTATION

- Arrow Functions verhalten sich hier anders
- Sie übernehmen `this` aus dem umgebenden Gültigkeitsbereich

```
1 function normalize () {
2   console.log(this.coords.map(n => n / this.length))
3 }
4
5 normalize.call({coords: [0, 2, 3], length: 5})
6 // → [0, 0.4, 0.6]
```

Speaker notes

Im Beispiel wird ausgenutzt, dass `this` in der Funktion in Pfeilnotation aus dem umgebenden Gültigkeitsbereich übernommen wird. So funktioniert es also nicht:

```
function normalize () {
  console.log(this.coords.map(function(n) { return n / this.length; }))
}
normalize.call({coords: [0, 2, 3], length: 5})
// → [ NaN, NaN, NaN ]
```

Da wir nicht den Strict Mode gesetzt haben, ist das zweite `this` das globale Objekt und `this.length` nicht definiert. Die Division führt zu NaN. In der Regel ist der Strict Mode empfehlenswert. Er hätte hier direkt zu einer Fehlermeldung geführt.

Um das Problem ohne Einsatz einer Arrow Function zu beheben, könnte man zu einem kleinen Trick greifen (ob die Variable `that` oder anders genannt wird, spielt hier keine Rolle):

```
"use strict"

function normalize () {
  let that = this
  console.log(this.coords.map(function(n) { return n / that.length; }))
}

normalize.call({coords: [0, 2, 3], length: 5})
// → [ 0, 0.4, 0.6 ]
```

Auch bei der abgekürzten Methodenschreibweise ist `this` das Objekt, für das die Methode aufgerufen wird. Hier sieht man den Unterschied zur Funktion in Pfeilnotation:

```
let cat = {
  type: "cat",
  say1() { return  "Meow from " + this.type },
  say2: () => "Meow from " + this.type,
}
console.log( cat.say1() )    /* → Meow from cat         */
console.log( cat.say2() )    /* → Meow from undefined   */
```

Damit haben wir die wichtigsten Aufrufvarianten und Belegungen von `this` beieinander. Was noch fehlt ist der Funktionsaufruf als Konstruktor, den wir gleich ansehen werden.

PROTOTYP

```
1 let empty = {}
2 console.log(empty.toString)          /* → [Function: toString]  */
3 console.log(empty.toString())        /* → [object Object]       */
```

- Wieso hat ein leeres Objekt eine Methode `toString`?
- Die meisten Objekte haben ein Prototyp-Objekt
- Dieses fungiert als Fallback für Attribute und Methoden
- Vererbung einmal anders...

PROTOTYP

```
> Object.getPrototypeOf({}) === Object.prototype
true

> Object.getOwnPropertyNames(Object.prototype)
[ 'constructor', 'hasOwnProperty', 'isPrototypeOf',
  'propertyIsEnumerable', 'toString', 'valueOf', ... ]
```

- Methoden und Attribute von `Object.prototype` sind auch für das leere Objekt { } verfügbar
- `toString` ist eine dieser Methoden

Speaker notes

`Object.prototype` enthält also unter anderem eine allgemeine `toString`-Methode. Vielleicht wundern Sie sich, dass `Object.prototype` selbst das leere Objekt ausgibt:

```
> Object.prototype
{}
```

Keine Spur von `toString`. Grund: nicht alle Attribute sind *enumerable*. Normalerweise werden nur die Attribute angezeigt, welche *enumerable* sind. `Object.getOwnPropertyNames` zeigt auch die anderen Attribute an. Attribute bieten somit eine Reihe weiterer Möglichkeiten als die, die wir bisher eingeführt haben. Interessierte können die Beschreibung von `Object.defineProperty()` hier nachlesen:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

Seit ES2022 gibt es auch eine Möglichkeit, das Vorhandensein eines bestimmten Attributs zu überprüfen:

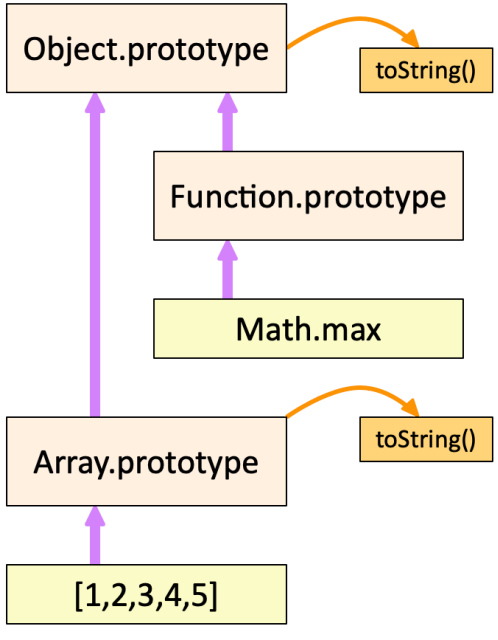
```
> Object.hasOwn(Object.prototype, 'toString')
true
```

PROTOTYP

- Funktionen haben `Function.prototype` als Prototyp
- Arrays haben `Array.prototype` als Prototyp
- Diese Prototypen haben `Object.prototype` als Prototyp

```
> Object.getPrototypeOf(Math.max) === Function.prototype
true

> Object.getPrototypeOf(Function.prototype) === Object.prototype
true

> Object.getPrototypeOf([]) === Array.prototype
true

> Object.getPrototypeOf(Array.prototype) === Object.prototype
true
```

PROTOTYPENKETTE




```
> [1,2,3,4,5].toString()
'1,2,3,4,5'

> Math.max.toString()
'function max() { [native code] }'

> Object.getOwnPropertyNames(Array.prototype)
['length', ... , 'toString']

> Object.getOwnPropertyNames(Object.prototype)
['constructor', ... , 'toString']
```

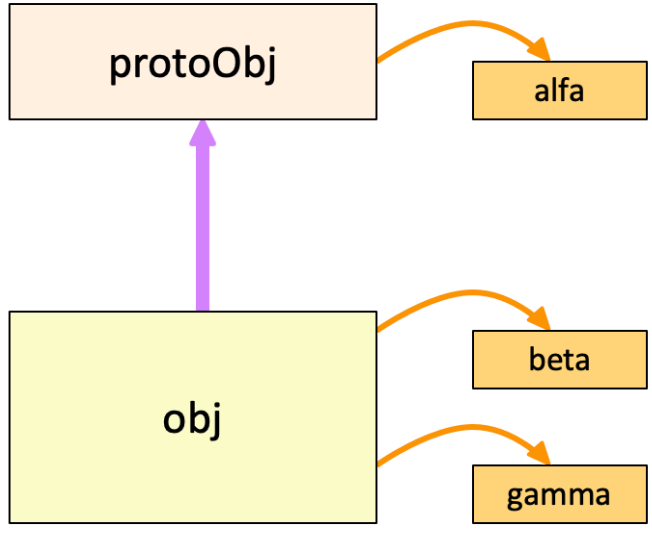
Speaker notes

Der Zugriff auf Attribute über Prototypen kann auch über mehrere Stufen erfolgen. Man spricht daher von der *Prototypen-Kette*. Wir haben also eine Art *Vererbung* über die Prototypen-Kette. Aber noch keine Klassen gesehen.

PROTOTYP

- Mit `Object.create` kann ein Objekt mit vorgegebenem Prototyp angelegt werden
- Es kann dann mit weiteren Attributen versehen werden

```
> let protoObj = { alfa: 1 }
> let obj = Object.create(protoObj)
> obj
{}

> obj.beta = 2
> obj.gamma = 3
> obj
{ beta: 2, gamma: 3 }

> obj.alfa
1
```



Speaker notes

Das Zuweisen der Attribute `beta` und `gamma` zu `obj` wäre auch so möglich:

```
> Object.assign(obj, {beta: 2, gamma: 3})
```

Das Beispiel zeigt, dass beim lesenden Zugriff auf das Attribut `alfa` von `obj` auf den Prototyp zugegriffen wird, da `obj` selbst kein Attribut `alfa` hat. Beim Schreiben wird aber nicht der Prototyp verändert:

```
> obj.alfa = 10
> obj
{ beta: 2, gamma: 3, alfa: 10 }
> protoObj
{ alfa: 1 }
```

WEITERES BEISPIEL

```
let protoRabbit = {
  speak (line) {
    console.log(`The ${this.type} rabbit says '${line}'`)
  }
}
let killerRabbit = Object.create(protoRabbit)
killerRabbit.type = "killer"
killerRabbit.speak("SKREEEE!")
// → The killer rabbit says 'SKREEEE!'
```

- Methode wird von `protoRabbit` genommen (geerbt)
- Variante zur Methodendefinition
  (statt: `speak: function (line) {...}`)

Speaker notes

Prototypen dienen also als Container für Attribute und Methoden, welche allen zugehörigen Objekten gemeinsam sind.

JSON

- Mit `JSON.stringify` werden Objekte serialisiert
- Methoden werden dabei nicht übernommen
- Prototyp wird ebenfalls nicht ins JSON übernommen
- Muss nach dem Parsen bei Bedarf wieder hergestellt werden

```
> let dataStrg  = '{"type":"cat","name":"Mimi","age":3}'
> let protoData = { category: "animal" }

> data = Object.assign(Object.create(protoData), JSON.parse(dataStrg))
{ type: 'cat', name: 'Mimi', age: 3 }
> data.category
'animal'
```

ÜBERSICHT

- Prototypen und this
- Konstruktoren und Vererbung
- Gewohntere Syntax: Klassen
- Test-Driven Development

OBJEKT MIT PROTOTYP

```
1 let protoPerson = {...}     /* Prototype */
2
3 function makePerson (name) {
4   let person = Object.create(protoPerson)
5   person.name = name
6   return person
7 }
```

- Objekt mit bestimmtem Prototyp erzeugen
- Dabei auch gleich Attribute belegen
- Das geht auch mit Hilfe von Konstruktoren...

KONSTRUKTOR

- Mit `function` definierte Funktionen können als Konstruktor interpretiert und mit `new` aufgerufen werden
- `this` ist dabei das neu angelegte Objekt
- Konvention: Konstruktoren mit grossen Anfangsbuchstaben

```
1 /* noch nicht ganz ideal, wird gleich verbessert... */
2 function Person (name) {
3   this.name = name
4   this.toString = function () {return `Person with name '${this.name}'`}
5 }
6
7 let p35 = new Person("John")
8 console.log(""+p35)   // → Person with name 'John'
```

Speaker notes

Funktionen, welche als Konstruktoren gedacht sind, sollten unbedingt mit grossem Anfangsbuchstaben geschrieben werden, damit klar ist, dass sie mit `new` aufgerufen werden müssen. Wird das `new` weggelassen, wird im Beispiel eine globale Variable `name` aber kein Objekt angelegt.

Das `""+p35` sorgt dafür, dass ein String erzeugt wird, was automatisch zum Aufruf von `toString()` führt. Man hätte natürlich die Methode auch direkt aufrufen können: `p35.toString()`.

Selbstverständlich ist *p35* kein guter Variablenname. Das soll hier nur zeigen, dass es sich um irgendeine Instanz von *Person* handelt.

Preisfrage: würde `toString` auch funktionieren, wenn es hier in Pfeilnotation definiert würde? Also so:

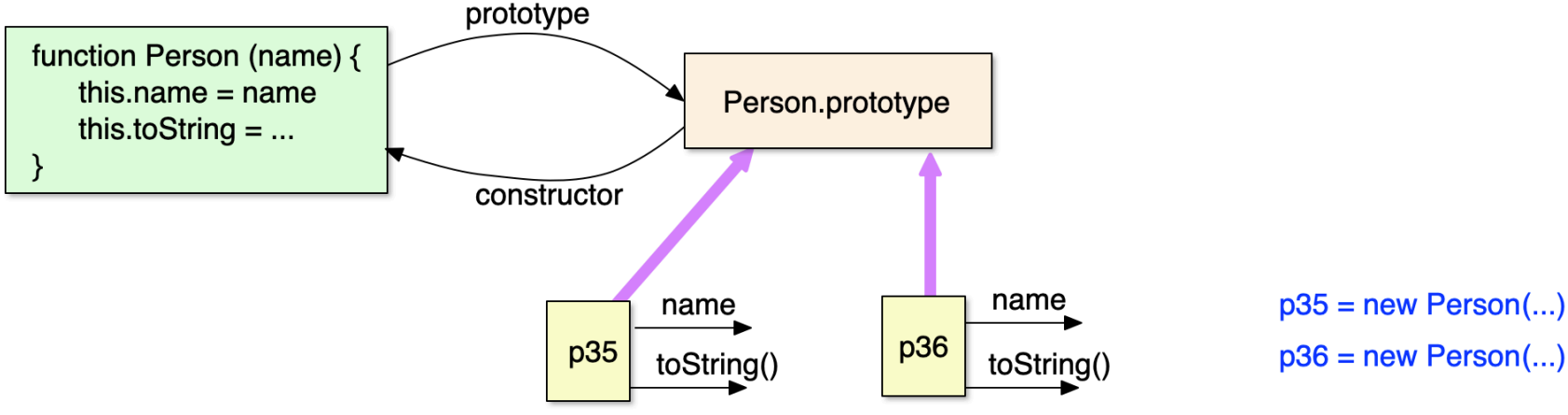
```
function Person (name) {
  this.name = name
  this.toString = () => `Person with name '${this.name}'`
}
```

In *Arrow Functions* wird `this` aus der Umgebung der Definition entnommen. Die ist hier aber ebenfalls das neu definierte Objekt, daher geht es auch so. Anders ist es, wenn toString später hinzugefügt wird:

```
// so geht's nicht:
p35.toString = () => `Person with name '${this.name}'`
console.log(p35.toString())
// → "Person with name 'undefined'"

// hier muss function verwendet werden, dann geht's:
p35.toString = function () {return "Person with name '" + this.name + "'"}
console.log(p35.toString())
// → "Person with name 'Mary'"
```

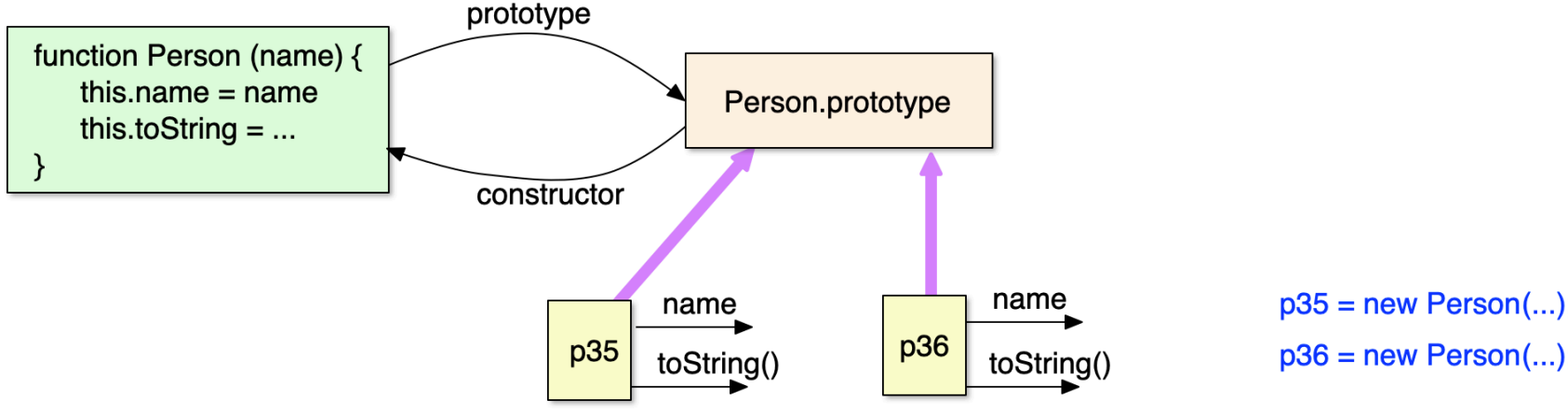
Keine Sorge, wenn Sie das nicht gleich nachvollziehen können. Das ist schon höhere JavaScript-Kunst...

KONSTRUKTOR



- Funktion hat `prototype`-Attribut: Referenz zu Prototyp
- Prototyp hat `constructor`-Attribut: zurück zur Funktion
- Objekte erben vom Prototyp, nicht vom Konstruktor

KONSTRUKTOR



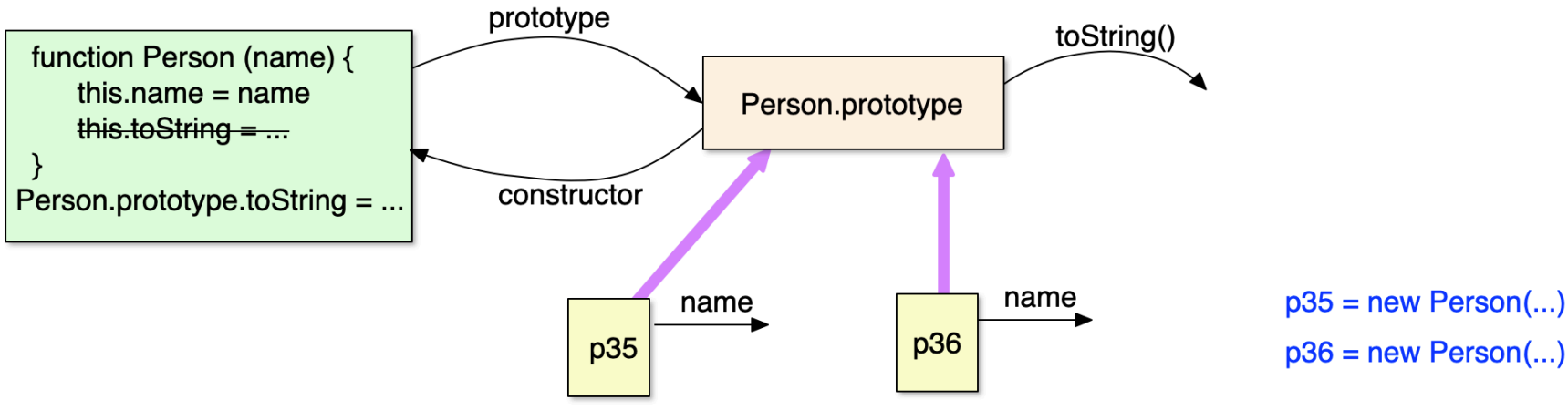

```
> Object.getPrototypeOf(p35) === Person.prototype
true

> Person.prototype.constructor === Person
true
```

PROTOTYP

- Im vorhergehenden Beispiel erhält jedes Objekt eine eigene `toString`-Methode, was unnötig ist
- Gemeinsame Attribute sollten im Prototyp angehängt werden

```
1 function Person (name) {
2   this.name = name
3 }
4 Person.prototype.toString = function () {
5   return `Person with name '${this.name}'`
6 }
7
8 let p35 = new Person("John")
```

PROTOTYP




```
> p35.toString()
Person with name 'John'

> Object.getOwnPropertyNames(p35)
[ 'name' ]

> p35 instanceof Person
true
```

Speaker notes

Dadurch dass sich `toString` nun zentral am Prototyp von `Person` befindet, kann es auch zentral für alle bereits mit `new Person` erzeugten Objekte geändert werden. Ausserdem kann `toString` lokal für einzelne Objekte überschrieben werden.

`Object.getOwnPropertyNames` listet die Namen aller Attribute, welche sich direkt im Objekt befinden. Geerbte Attribute tauchen hier nicht auf.

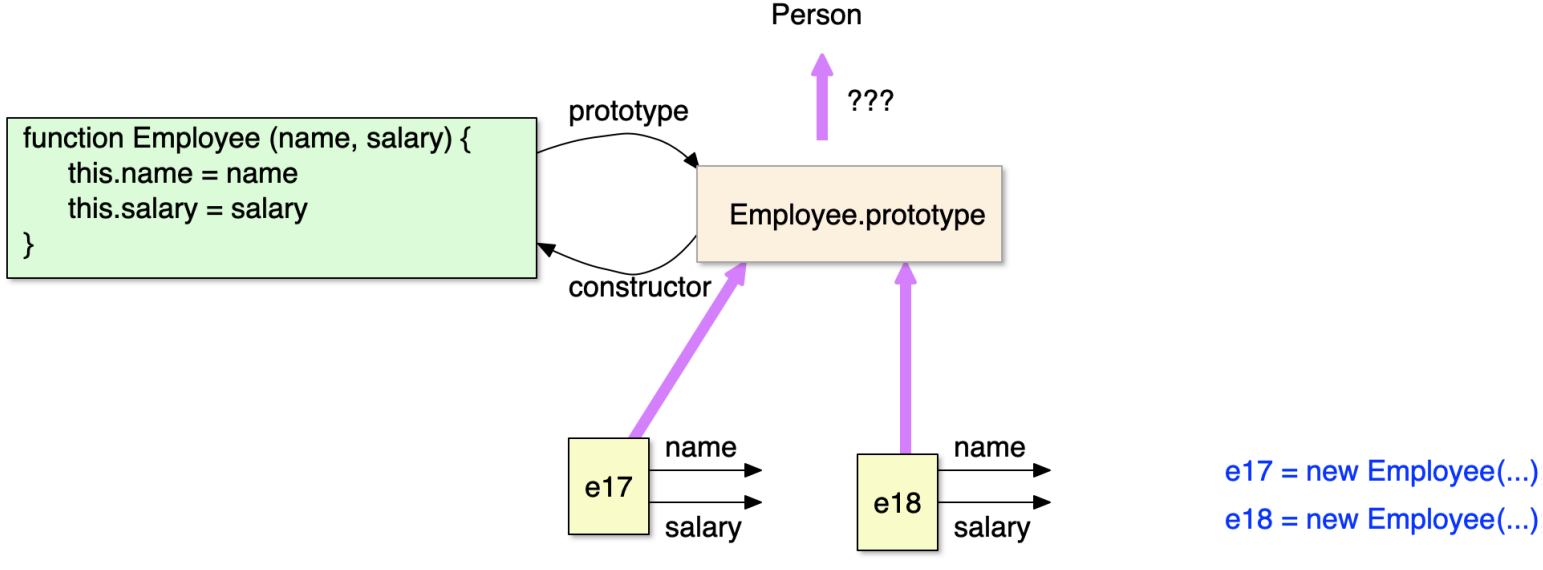
Der `instanceof`-Operator liefert true, wenn der Prototyp des Konstruktors sich in der Vererbungskette des Objekts befindet.

Als Funktion könnte dies etwa so implementiert werden:

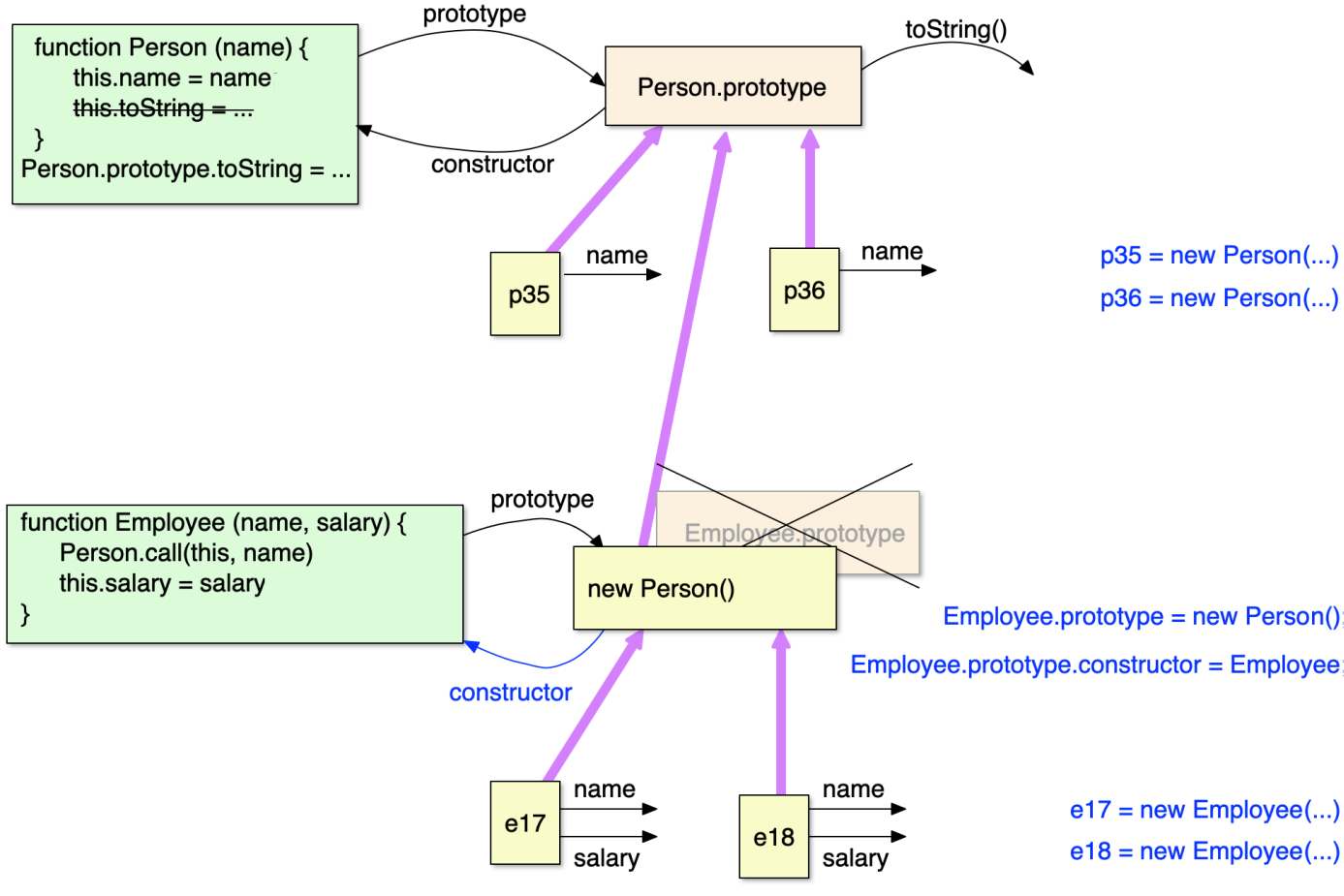
```
function instanceOf (obj, constr) {
  let curr = Object.getPrototypeOf(obj)
  if (["number", "string", "boolean"].includes(typeof obj)) return false
  while (curr) {
    if (curr === constr.prototype) return true
    else curr = Object.getPrototypeOf(curr)
  }
  return false
}

> instanceOf([], Array)
true
> instanceOf([], Object)
true
> instanceOf(Math.max, Function)
true
> instanceOf(12, Number)
false
```

PROTOTYPEN-KETTE

- Ein Objekt erbt vom Prototyp seines Konstruktors
- Möglich: Prototyp durch Objekt eines anderen Konstruktors ersetzen
- Dadurch kann eine Vererbungshierarchie aufgebaut werden



PROTOTYPEN-KETTE



PROTOTYPEN-KETTE

```
 1 function Employee (name, salary) {
 2   Person.call(this, name)
 3   this.salary = salary
 4 }
 5
 6 Employee.prototype = new Person()
 7 Employee.prototype.constructor = Employee
 8
 9 let e17 = new Employee("Mary", 7000)
10
11 console.log(e17.toString())    /* → Person with name 'Mary' */
12 console.log(e17.salary)        /* → 7000 */
```

Speaker notes

Der `super`-Konstruktor wird folgendermassen aufgerufen:

```
Person.call(this, name)
```

Grund: das neue Objekt ist Ziel der Attribute, auch der Attribute, welche vom Super-Konstruktor angelegt werden. Alternativ könnte statt `call` auch dafür gesorgt werden, dass `this` durch *Method Invocation* korrekt gesetzt wird:

```
function Employee (name, salary) {
  this.base = Person
  this.base(name)
  this.salary = salary
}
```

Auf diese Weise erhält jedes neue Objekt noch ein Attribut `base`, eine Referenz zur Basisklasse (eigentlich: Basis-Konstruktor).

PROTOTYPENKETTE

- Lesender Zugriff:
  Wenn Attribut nicht vorhanden ist, wird es entlang der Prototypenkette gesucht
- Schreibender Zugriff:
  Attribut wird direkt im Objekt angelegt

- Objekt kann auch keinen Prototyp haben (`null` setzen)
- Für die meisten Objekte steht `Object.prototype` am Ende der Prototypenkette

ÜBERSICHT

- Prototypen und this
- Konstruktoren und Vererbung
- Gewohntere Syntax: Klassen
- Test-Driven Development

KLASSEN

- Vererbung über Prototypen ist gewöhnungsbedürftig
- Wenn auch sehr mächtig: damit lassen sich verschiedene Varianten von Objektorientierung umsetzen
- ES6: Klassen eingeführt
- Syntax eher an andere OOP-Sprachen angelehnt
- Letztlich nur *Syntactic Sugar* für Prototypensystem

Speaker notes

Hinter den Klassen versteckt sich also das System basierend auf Objekten und ihren Prototypen. Auch wenn es seit ES6 Klassen gibt, muss man als JavaScript-Entwickler die Prototypen verstehen, denn das ist der eigentliche Mechanismus hinter den Klassen. Ausserdem sind die Ausdrucksmöglichkeiten mit Prototypen vielseitiger und man findet man zahlreiche Beispiele in Büchern, Blog-Beiträten und Bibliotheks-Dokumentationen, welche das Verständnis der Prototypen voraussetzen.

KLASSEN

```
1  class Person {
2    constructor (name) {
3      this.name = name
4    }
5    toString () {
6      return `Person with name '${this.name}'`
7    }
8  }
9
10 let p35 = new Person("John")
11 console.log(p35.toString())    // → Person with name 'John'
```

Speaker notes

Der eigentliche Konstruktor ist nun eine Methode mit dem Namen *constructor*. Sie wird an den Klassennamen gebunden. Weitere Methoden werden an den Prototyp angehängt.

KLASSEN: VERERBUNG

```
1  class Employee extends Person {
2    constructor (name, salary) {
3      super(name)
4      this.salary = salary
5    }
6    toString () {
7      return `${super.toString()} and salary ${this.salary}`
8    }
9  }
10
11 let e17 = new Employee("Mary", 7000);
12
13 console.log(e17.toString()) /* → Person with name 'Mary' and salary 7000 */
14 console.log(e17.salary)     /* → 7000 */
```

KLASSEN: GETTER UND SETTER

```
1  class PartTimeEmployee extends Employee {
2    constructor (name, salary, percentage) {
3      super(name, salary)
4      this.percentage = percentage
5    }
6    get salary100 () { return this.salary * 100 / this.percentage }
7    set salary100 (amount) { this.salary = amount * this.percentage / 100 }
8  }
9
10 let e18 = new PartTimeEmployee("Bob", 4000, 50)
11
12 console.log(e18.salary100)   /* → 8000 */
13 e18.salary100 = 9000
14 console.log(e18.salary)      /* → 4500 */
```

Speaker notes

Getter und Setter sind nicht auf Klasssen beschränkt, sondern auch in Objekten möglich:

```
const language = {
  set current(name) {
    this.log.push(name);
  },
  log: []
};

language.current = 'EN';
language.current = 'FA';

console.log(language.log);
// expected output: Array ["EN", "FA"]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set

Wir gehen hier nicht weiter ins Detail mit den Klassen in JavaScript. In neueren JavaScript-Versionen hat es hier diverse Änderungen und Erweiterungen gegeben. So gibt es mittlerweile auch statische Variablen und Methoden und seit ES2022 auch private Variablen und Methoden:

```
class Person {
  // instance private field
  #firstName;
  constructor(firstName) {
    this.#firstName = firstName;
  }
  // method
  describe() {
    return `Person named ${this.#firstName}`;
  }
  // static method
  static extractNames(persons) {
    return persons.map(person => person.#firstName);
  }
}
```

https://exploringjs.com/impatient-js/ch_classes.html

ÜBERSICHT

- Prototypen und this
- Konstruktoren und Vererbung
- Gewohntere Syntax: Klassen
- Test-Driven Development

TEST-DRIVEN DEVELOPMENT, TDD

- Tests konsequent vor den zu testenden Komponenten erstellt
- Häufig bei der agilen Software-Entwicklung eingesetzt
- Verbessert Verständnis der zu erstellenden Komponenten
- Tests als Spezifikation für korrektes Verhalten der Software
- Refactoring erleichtert

„I like test-driven development as a methodology but I hate it as a religion.”
Douglas Crockford, FullStack London 2018

JASMINE

> *„Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests. “*

https://jasmine.github.io/index.html

JASMINE

- Testsuite besteht aus mehreren Specs
- Ziel in natürlicher Sprache beschrieben
- Suites und Specs sind Funktionen
- Für Node.js ebenso wie für Browser-Umgebung

```
describe("A suite is just a function", function () {
  let a

  it("and so is a spec", function () {
    a = true
    expect(a).toBe(true)
  })
})
```

JASMINE INSTALLATION

```
$ npm init
$ npm install --save-dev jasmine
$ npx jasmine init
$ npx jasmine examples
```

```
jasmine
├── lib
│   └── jasmine_examples
│       ├── Player.js
│       └── Song.js
├── node_modules
├── package-lock.json
├── package.json
└── spec
    ├── helpers
    │   └── jasmine_examples
    │       └── SpecHelper.js
    ├── jasmine_examples
    │   └── PlayerSpec.js
    └── support
        └── jasmine.json
```

- Legt Projekt mit lokal installiertem Jasmine an
- Kopiert ein paar Beispiel-Dateien ins Projekt
- Konfiguration in `spec/support/jasmine.json`

https://jasmine.github.io/setup/nodejs.html

BEISPIEL (PROGRAMMLOGIK)

```
1  /* Player.js */
2  function Player() {
3  }
4  Player.prototype.play = function(song) {
5    this.currentlyPlayingSong = song
6    this.isPlaying = true
7  }
8  Player.prototype.pause = function() {
9    this.isPlaying = false
10 }
11 Player.prototype.resume = function() {
12   if (this.isPlaying) {
13     throw new Error("song is already playing")
14   }
15   this.isPlaying = true
16 }
17 Player.prototype.makeFavorite = function() {
18   this.currentlyPlayingSong.persistFavoriteStatus(true)
19 }
20 module.exports = Player
```

Speaker notes

```
/* Song.js */
function Song() {
}
Song.prototype.persistFavoriteStatus = function(value) {
  // something complicated
  throw new Error("not yet implemented");
};
module.exports = Song;
```

BEISPIEL (ZUGEHÖRIGE TESTS)

```
1  /* PlayerSpec.js - Auszug */
2  describe("when song has been paused", function() {
3    beforeEach(function() {
4      player.play(song)
5      player.pause()
6    })
7
8    it("should indicate that the song is currently paused", function() {
9      expect(player.isPlaying).toBeFalsy()
10
11     /* demonstrates use of 'not' with a custom matcher */
12     expect(player).not.toBePlaying(song)
13   })
14
15   it("should be possible to resume", function() {
16     player.resume()
17     expect(player.isPlaying).toBeTruthy()
18     expect(player.currentlyPlayingSong).toEqual(song)
19   })
20 })
```

JASMINE: TESTS DURCHFÜHREN

```
$ npx jasmine
Randomized with seed 03741
Started
.....
5 specs, 0 failures
Finished in 0.014 seconds
Randomized with seed 03741 (jasmine --random=true --seed=03741)
```

Speaker notes

Und im Fehlerfall:

```
$ npx jasmine
Randomized with seed 09186
Started
....F

Failures:
1) Player when song has been paused should be possible to resume
  Message:
    Expected false to be truthy.
  Stack:
    Error: Expected false to be truthy.
        at <Jasmine>
        at UserContext.<anonymous> (/Users/.../spec/jasmine_examples/PlayerSpec
        at <Jasmine>



5 specs, 1 failure
Finished in 0.011 seconds
Randomized with seed 09186 (jasmine --random=true --seed=09186)
```

JASMINE: MATCHER

```
expect([1, 2, 3]).toEqual([1, 2, 3])
expect(12).toBeTruthy()
expect("").toBeFalsy()
expect("Hello planet").not.toContain("world")
expect(null).toBeNull()
expect(8).toBeGreaterThan(5)
expect(12.34).toBeCloseTo(12.3, 1)
expect("horse_ebooks.jpg").toMatch(/\w+.(jpg|gif|png|svg)/i)
...
```

JASMINE: MEHR

- Verhalten von Methoden oder ganzen Objekten simulieren
- Erstellen von Mock Objects mit Jasmine Spies

```
spyOn(dictionary, "hello")
expect(dictionary.hello).toHaveBeenCalled()

// oder...
spyOn(dictionary, "hello").and.returnValue("bonjour")
spyOn(dictionary, "hello").and.callFake(fakeHello)
```

JASMINE IM BROWSER

- Standalone Release herunterladen
  https://github.com/jasmine/jasmine/releases
- Beispiel-Quellen und -Tests ersetzen
- `SpecRunner.html`
  - anpassen (Quellen, Tests)
  - im Browser öffnen

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript
  https://eloquentjavascript.net/
- Ältere Slides aus WEB2 und WEB3
- Dokumentationen, u.a. zu Node.js, Jasmine

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 4 und 6 von:
  Marijn Haverbeke: Eloquent JavaScript
  https://eloquentjavascript.net/