WBE: JAVASCRIPT

PROTOTYPEN VON OBJEKTEN

ÜBERSICHT

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

ÜBERSICHT

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

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 Node-Modul ist `this` auf oberster Ebene an `module.exports` gebunden Das können Sie verifizieren, indem Sie die folgende Zeilen in einem Node-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 Node-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 `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 ]
```

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:

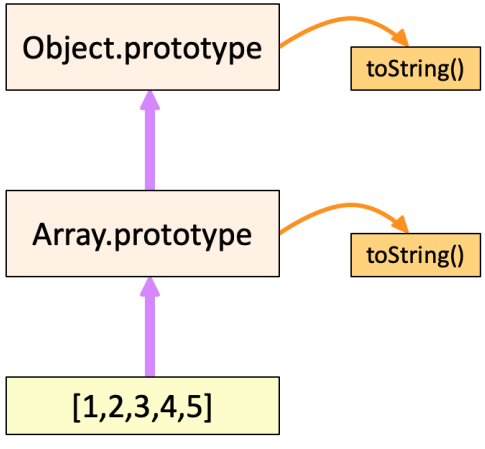
```
> Object.hasOwn(Object.prototype, 'toString')
true
```

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']
```

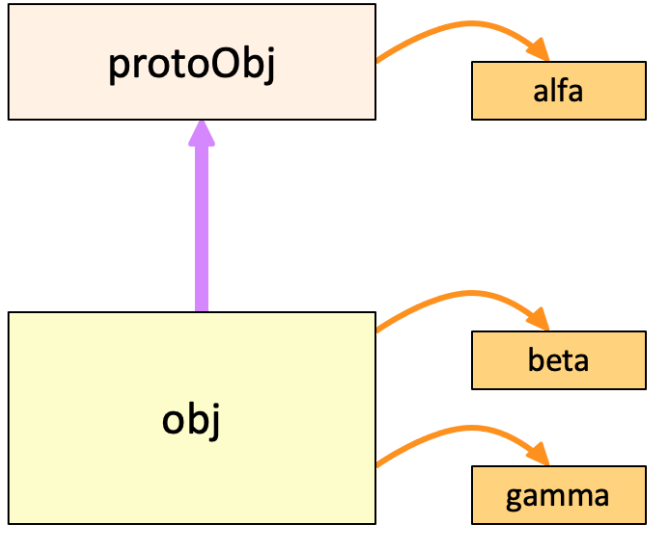
Speaker notes

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

- Funktionen können mit `new` aufgerufen werden
- In diesem Fall werden sie als Konstruktor interpretiert
- `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()`.

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

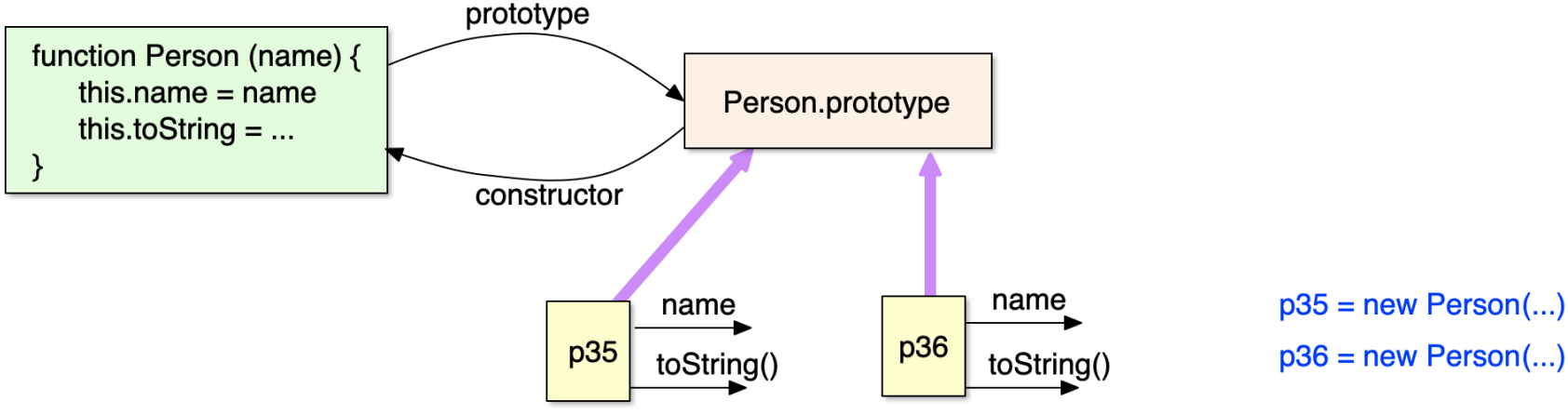
```
function Person (name) {
  this.name = name
  this.toString = () => `Person with name '${this.name}'`
}
```

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'"
```

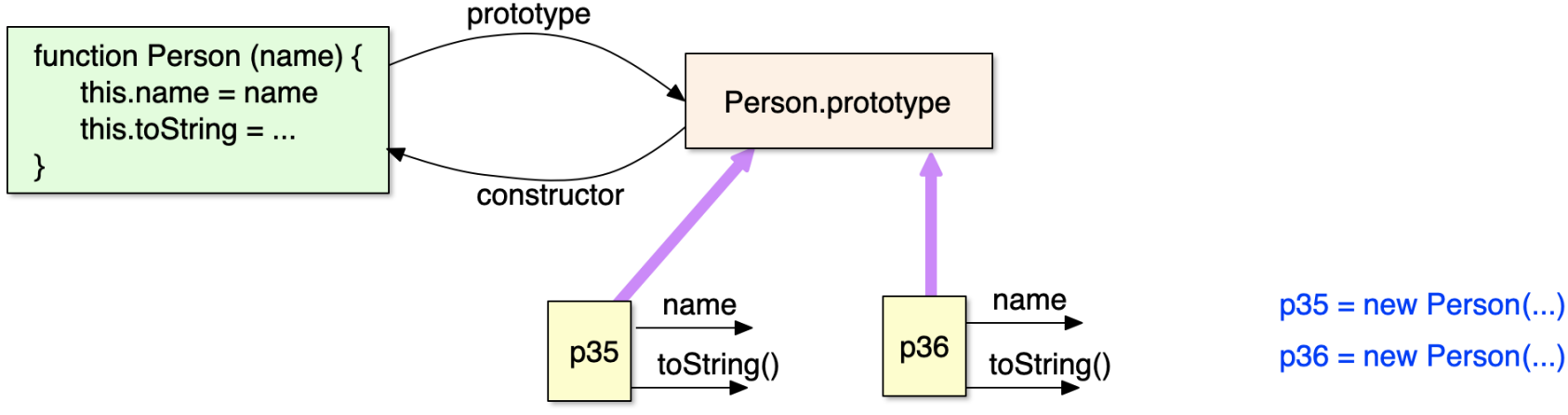
Keine Sorge, wenn Sie das nicht gleich nachvollziehen können. Das ist schon höhere JavaScript-Kunst...

KONSTRUKTOR



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

KONSTRUKTOR



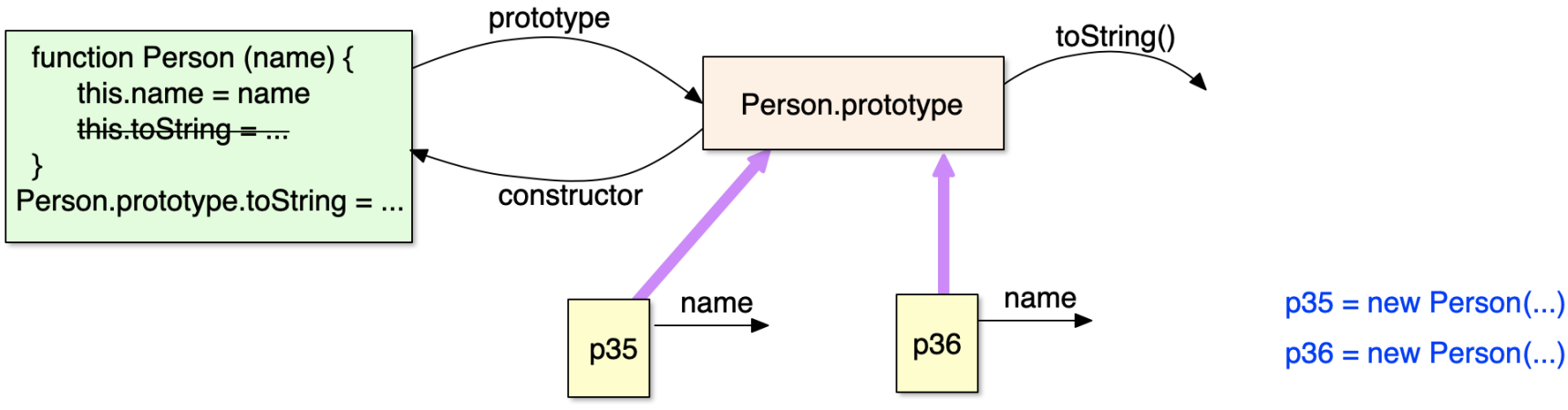

```
> Object.getPrototypeOf(p35) === Person.prototype
true

> 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.

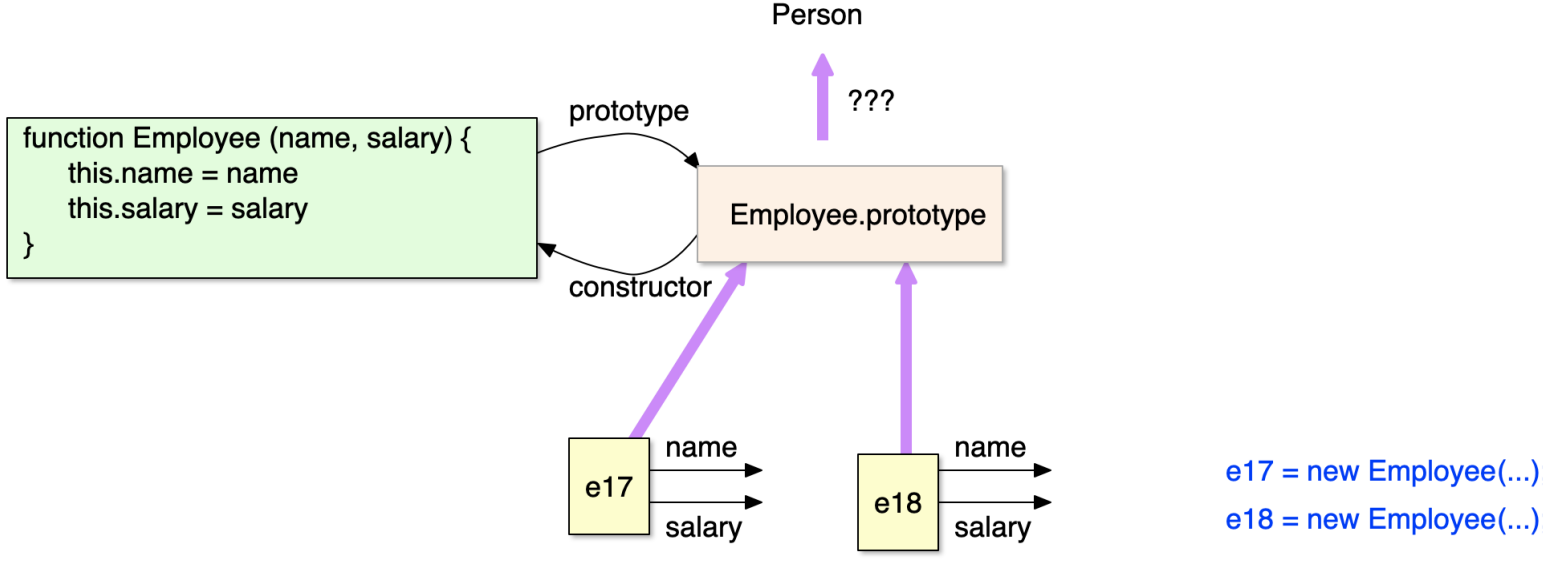
Der `instanceof`-Operator liefert true, wenn der Prototyp des Konstruktors sich in der Vererbungskette des Objekts befindet.

Als Funktion könnte dies etwa so implementiert werden:

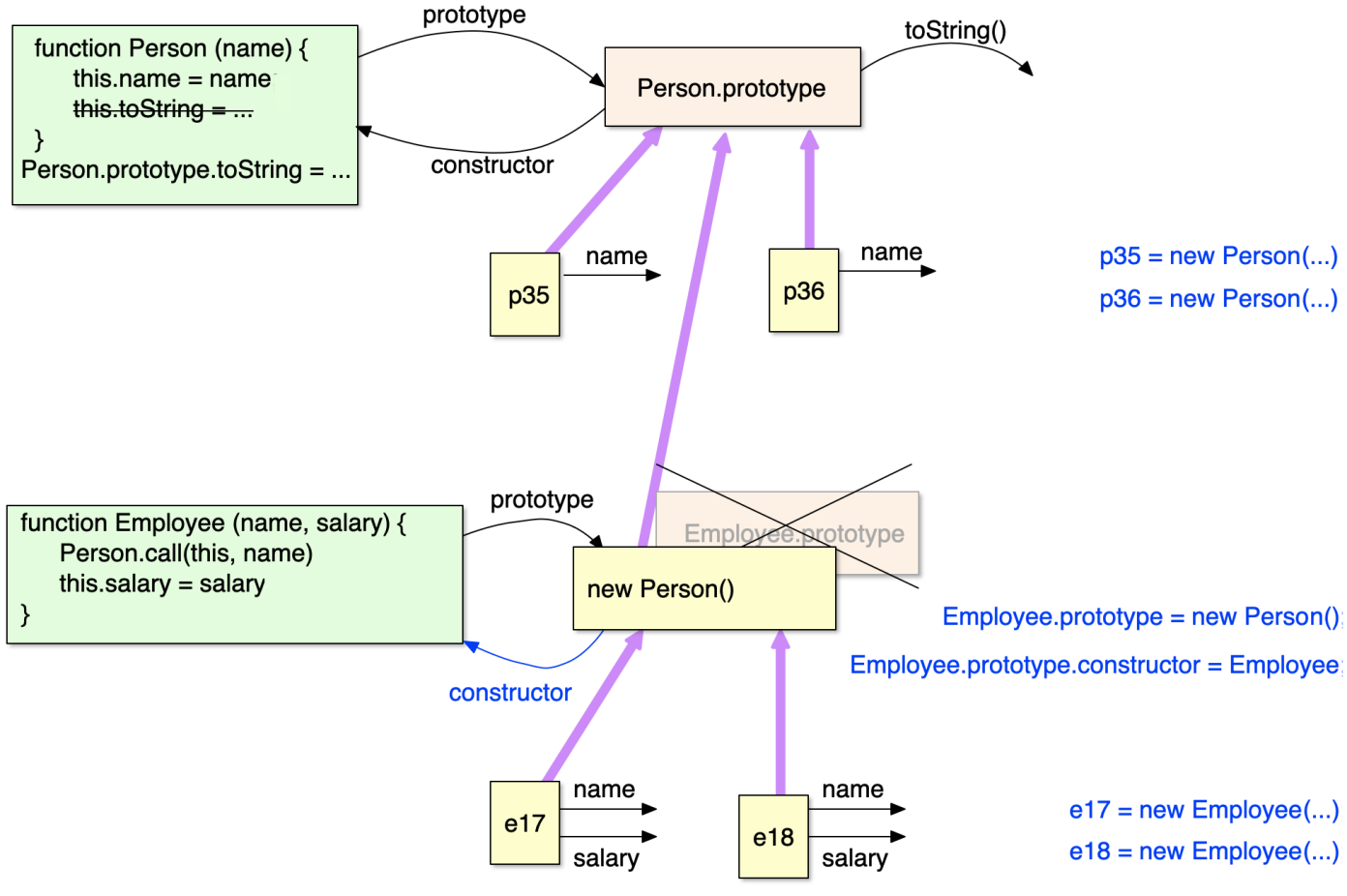
```
function instanceOf (obj, constr) {
  let curr = Object.getPrototypeOf(obj)
  if (["number", "string", "boolean"].includes(typeof obj)) return false
  while (curr) {
    if (curr === constr.prototype) return true
    else curr = Object.getPrototypeOf(curr)
  }
  return false
}

> 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, 3rd Edition
  https://eloquentjavascript.net/
- Ältere Slides aus WEB2 und WEB3
- Dokumentationen, u.a. zu Node.js, Jasmine

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 6 von:
  Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/