

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

Note:

Was das globale Objekt ist, ist abhängig von der Laufzeitumgebung. Im Browser ist das globale Objekt `window`. Es repräsentiert das aktuell geöffnete

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
```

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!'
```


bind

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

```
1 function speak (line) {  
2   console.log(`The ${this.type} rabbit says '${line}'`)  
3 }  
4 let hungryRabbit = {type: "hungry"}  
5  
6 let boundSpeak = speak.bind(hungryRabbit)  
7 boundSpeak("Burp!")  
8 // → 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]
```

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

PROTOTYP

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

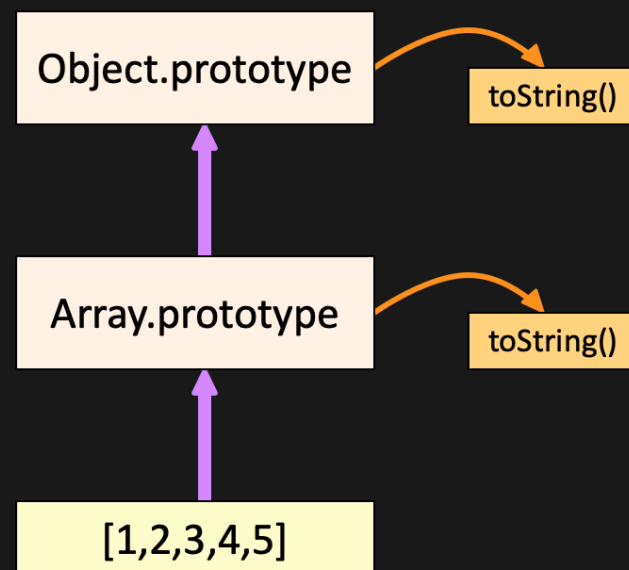
```
> Object.getPrototypeOf(Math.max) == Function.prototype  
true
```

```
> 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] }'
```

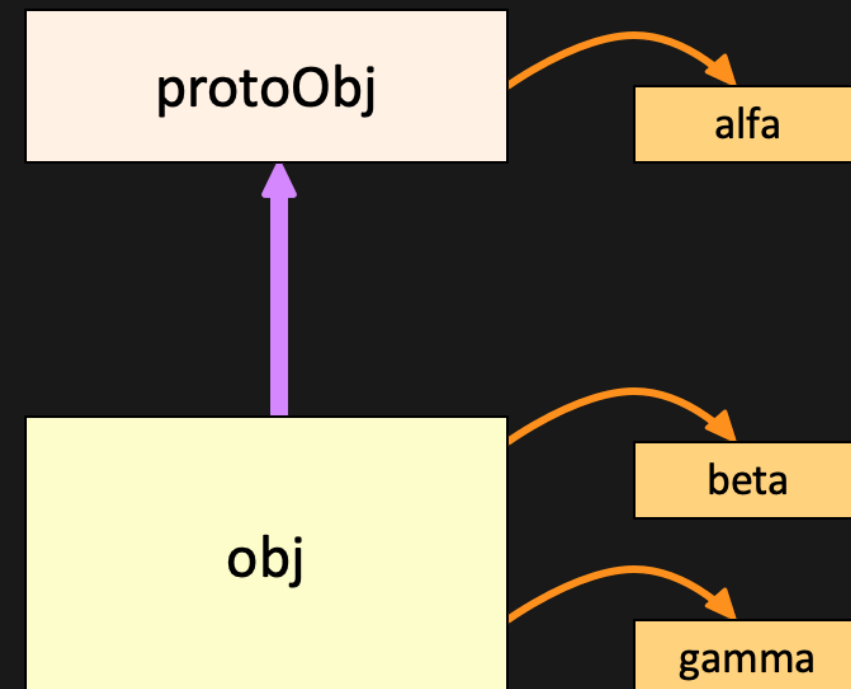
```
> Object.getOwnPropertyNames(Array.prototype)  
['length', ... , 'toString']
```

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

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
```



WEITERES BEISPIEL

```
1 let protoRabbit = {
2   speak (line) {
3     console.log(`The ${this.type} rabbit says '${line}'`)
4   }
5 }
6 let killerRabbit = Object.create(protoRabbit)
7 killerRabbit.type = "killer"
8 killerRabbit.speak("SKREEEE!")
9 // → The killer rabbit says 'SKREEEE!'
```

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

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 }
```

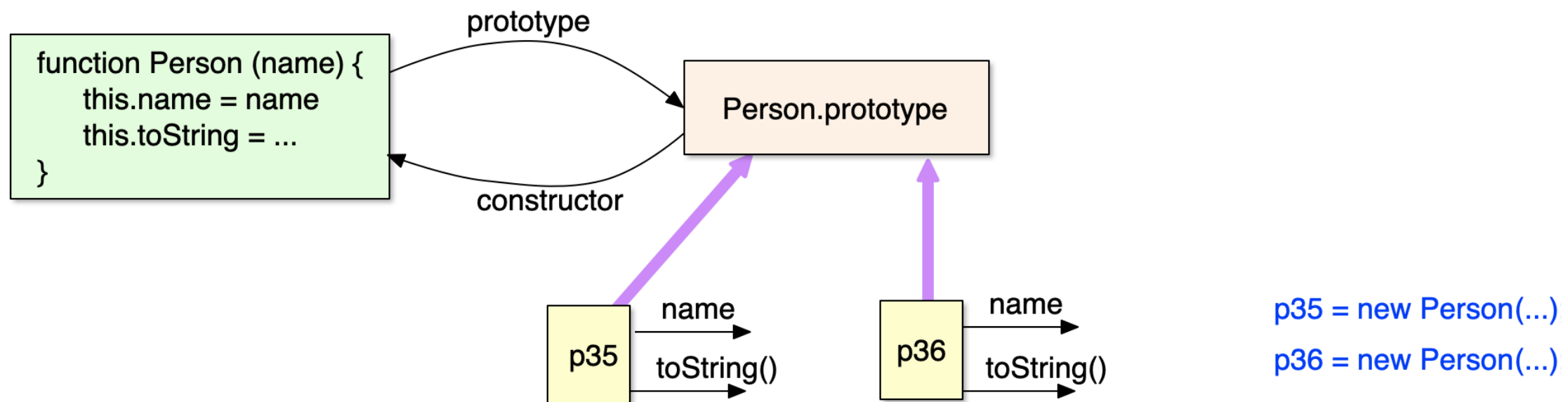
- Objekt mit bestimmtem Prototyp erzeugen
- Dabei auch gleich Attribute belegen
- Das geht auch mit Hilfe von Konstruktoren...

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

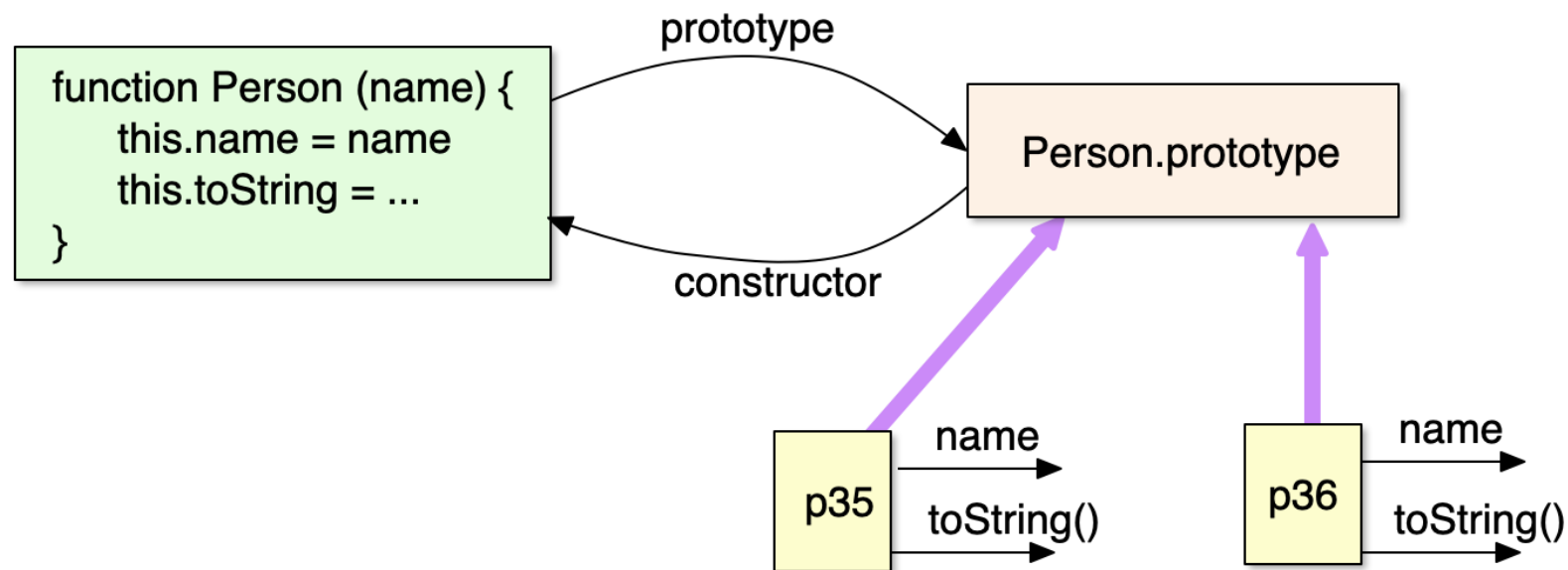
```
1 /* noch nicht ganz ideal, wird gleich verbessert... */
2 function Person (name) {
3     this.name = name
4     this.toString = function () {return `Person with name '${this.name}'`}
5 }
6
7 let p35 = new Person("John")
8 console.log(""+p35) // → Person with name 'John'
```

KONSTRUKTOR



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

KONSTRUKTOR



`p35 = new Person(...)`

`p36 = new Person(...)`

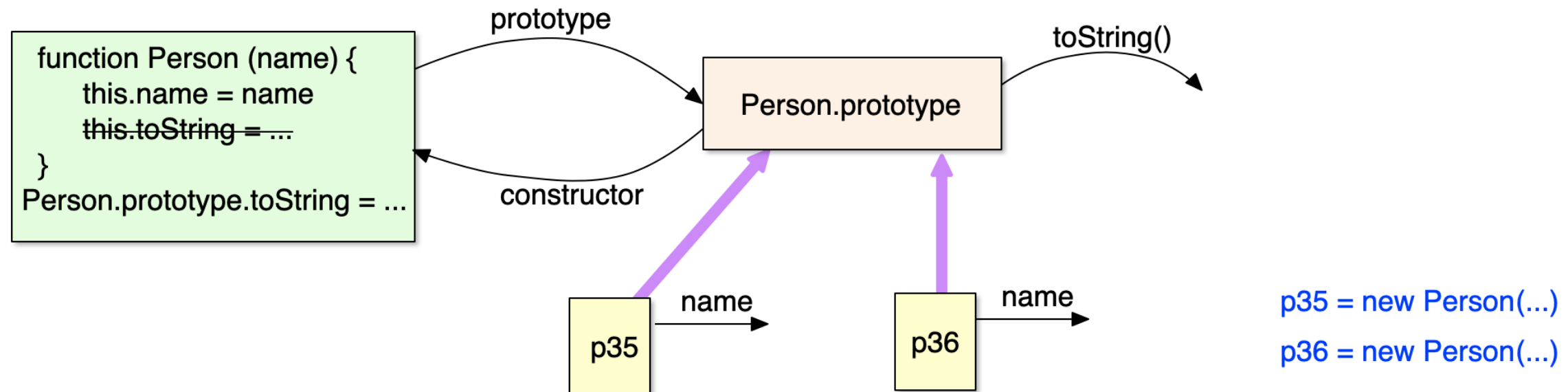
```
> Object.getPrototypeOf(p35) === Person.prototype  
true  
  
> Person.prototype.constructor === Person  
true
```

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")
```

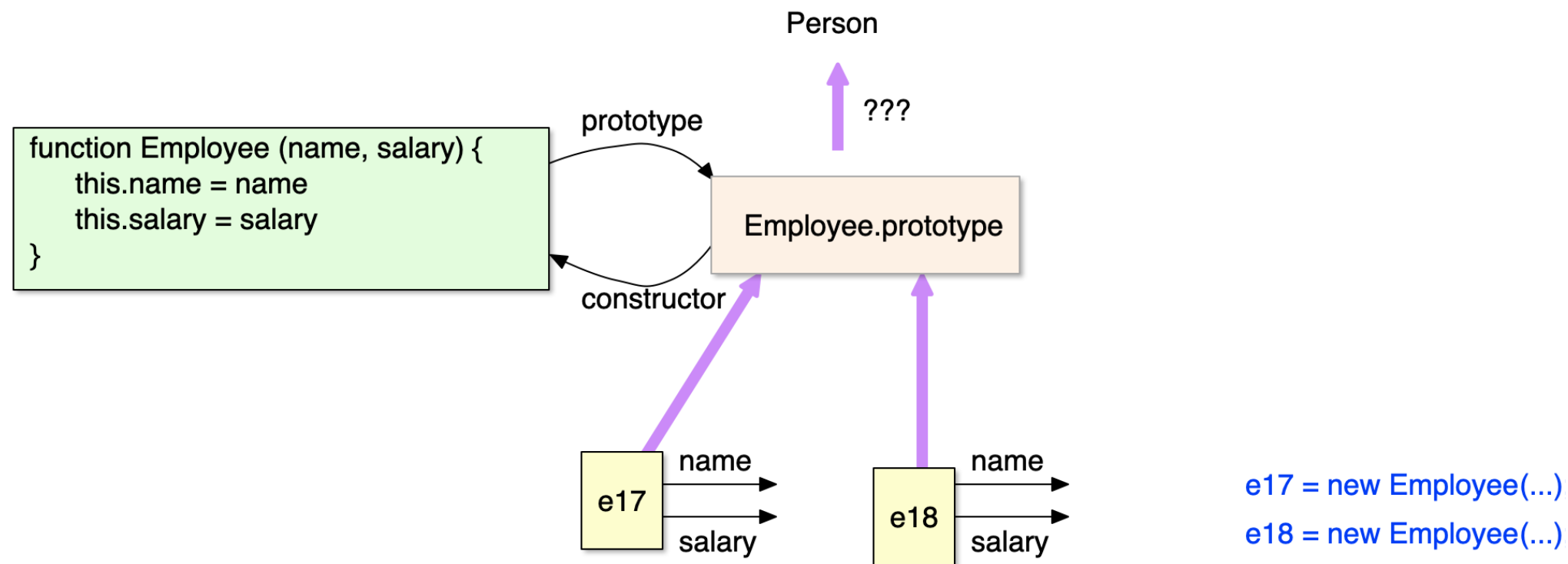
PROTOTYP



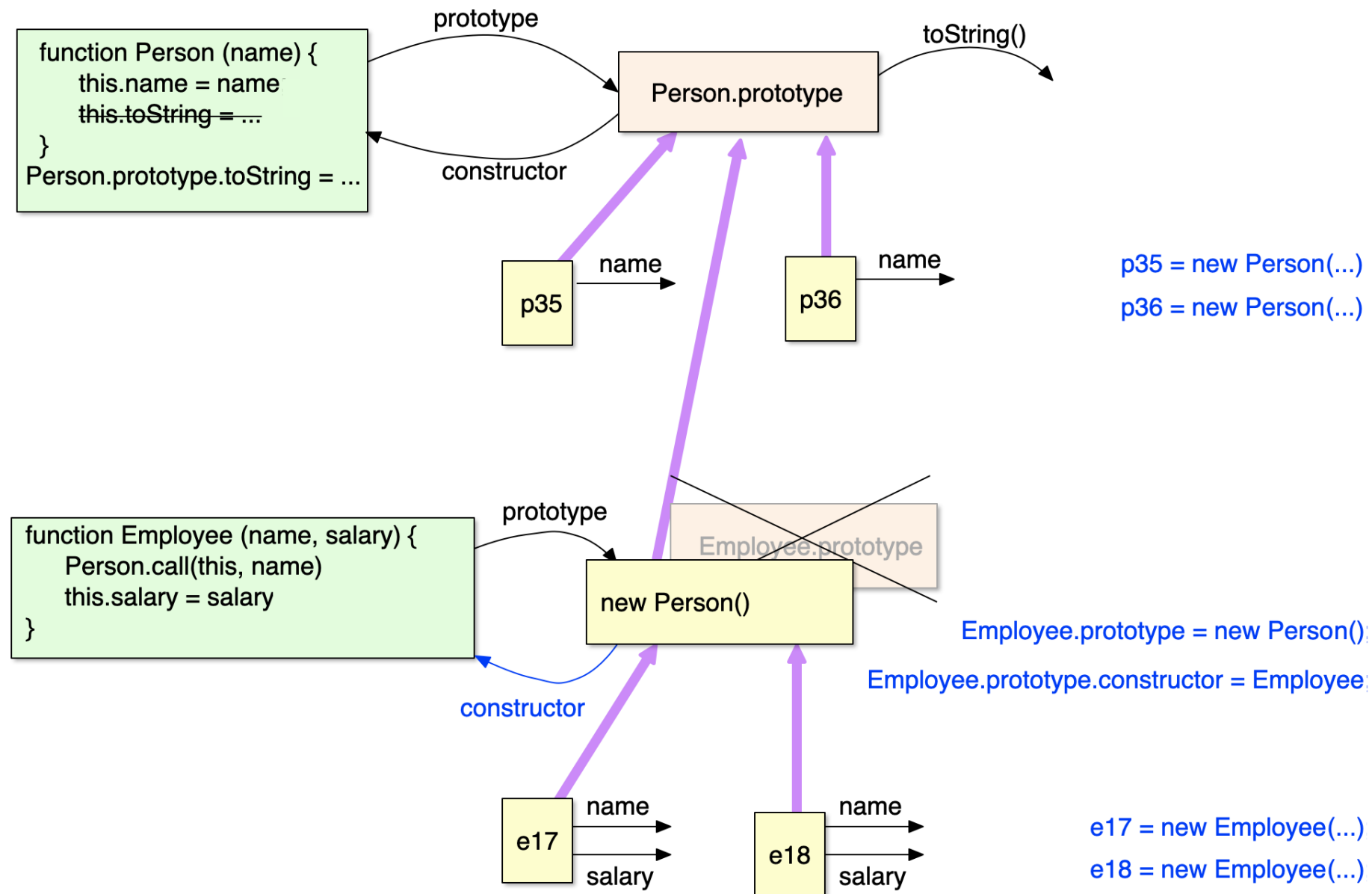
```
> p35.toString()  
Person with name 'John'  
  
> Object.getOwnPropertyNames(p35)  
[ 'name' ]  
  
> p35 instanceof Person  
true
```


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 */
```

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

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'
```

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 */
```

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
```

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)
```

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:
Marius Haverbeke: Eloquent JavaScript, 3rd Edition
<https://eloquentjavascript.net/>

