Disclaimer

Dies ist der Vortrag zum Seminar
"1919 – Seminar Moderne Programmiertechniken und"
im Wintersemester 2018/2019 an der FernUniversität Hagen.
Es ist die Präsentation der Ergebnisse aus der Seminararbeit mit dem Titel "Borrow – Inherit – Mix: Code-Wiederverwendung in JavaScript".

Eine etwas ausführlichere Version des Vortrags wurde im Rahmen des HannoverJS-User Group Meetings am 26.3.2019 vorgestellt.
Diese ausführlichere Version ist zusammen mit der Seminararbeit verfügbar unter
`https://github.com/opt12/BorrowInheritMix`
Es gilt die Lizenz:
Creative Commons Attribution Share Alike 4.0

Dies ist der Vortrag zum Seminar
"1919 – Seminar Moderne Programmiertechniken und"
im Wintersemester 2018/2019 an der FernUniversität Hagen.
Es ist die Präsentation der Ergebnisse aus der Seminararbeit mit dem Titel "Borrow – Inherit – Mix: Code-Wiederverwendung in JavaScript".

Eine etwas ausführlichere Version des Vortrags wurde im Rahmen des HannoverJS-User Group Meetings am 26.3.2019 vorgestellt.
Diese ausführlichere Version ist zusammen mit der Seminararbeit verfügbar unter
https://github.com/opt12/BorrowInheritMix
Es gilt die Lizenz:
Creative Commons Attribution Share Alike 4.0

Borrow – Inherit – Mix: Code-Wiederverwendung in JavaScript

Felix Eckstein

Student im Master Informatik

Talk im Rahmen von HannoverJS
26. März 2019

Borrow – Inherit – Mix:
Code-Wiederverwendung in JavaScript

Felix Eckstein

Student im Master Informatik

Talk im Rahmen von HannoverJS
26. März 2019

- Wer von euch programmiert in JavaScript?
- Wer von Euch ist der Meinung, dass JavaScript eine ernstzunehmende Programmiersprache ist?
-
- Es ist inzwischen eine richtige Sprache
- wir betrachten einige Features, die Programmieren *im Großen* ermöglichen
-
- Javascript ist –im Gegensatz zu vielen anderen Sprachen– Prototyp-basiert

Outline

Einführung
JavaScript: Wichtige Grundlagen
Objekterzeugung
Prototype-Chain
this-Binding
dynamische Objekte
Code-Reuse
Method Borrowing
Delegation und Vererbung
Mixins für orthogonalen Code-Resue
Functional Mixins
Fazit
Literatur

1. Einführung
2. JavaScript: Wichtige Grundlagen
   - Objekterzeugung
   - Prototype-Chain
   - `this`-Binding
   - dynamische Objekte
3. Code-Reuse
   - Method Borrowing
   - Delegation und Vererbung
   - Mixins für orthogonalen Code-Resue
   - Functional Mixins
4. Fazit


Einführung
JavaScript: Wichtige Grundlagen
Objekterzeugung
Prototype-Chain
this-Binding
dynamische Objekte
Code-Reuse
Method Borrowing
Delegation und Vererbung
Mixins für orthogonalen Code-Resue
Functional Mixins
Fazit

Outline

Code Wiederverwendung
Überblick (1/2)

- Code Wiederverwendung ist ein zentrales Thema der SW-Entwicklung
- Ohne Wiederverwendung von Code ist die Komplexität nicht beherrschbar
- *globale* Methoden zur Wiederverwendung
  - Libraries
  - Frameworks
  - Modulsysteme

Das Augenmerk dieser Arbeit liegt auf *lokalen* Methoden, wie z. B. innerhalb einer Applikation Code Wiederverwendung stattfinden kann.

- Code Wiederverwendung ist ein zentrales Thema der SW-Entwicklung
- Ohne Wiederverwendung von Code ist die Komplexität nicht beherrschbar
- globale Methoden zur Wiederverwendung
  - Libraries
  - Frameworks
  - Modulsysteme

Das Augenmerk dieser Arbeit liegt auf lokalen Methoden, wie z. B. innerhalb einer Applikation Code Wiederverwendung stattfinden kann.

- Wiederverwendung ist notwendig um die Komplexität zu behrrschen.
- *globale* Methoden zur Wiederverwendung
  - Libraries
  - Frameworks
  - Modulsysteme

Code Wiederverwendung
Überblick (2/2)

Am Beispiel der Sprache JavaScript werden folgende Methoden vorgestellt:

- Method Borrowing
- (prototypical) Inheritance
- Mixins
- functional Mixins

Von der Betrachtung ausgenommen sind die Methoden der *klassischen* Vererbung, Modulsysteme und Frameworks.

Am Beispiel der Sprache JavaScript werden folgende Methoden vorgestellt:
- Method Borrowing
- (prototypical) Inheritance
- Mixins
- functional Mixins

Von der Betrachtung ausgenommen sind die Methoden der *klassischen* Vererbung, Modulsysteme und Frameworks.

- Die vorgestellten Methoden bauen auf dem *prototypischen* Objektsystem von JS auf
  - Method Borrowing
  - (prototypical) Inheritance
  - Mixins
  - functional Mixins
- Diese Methoden betreffen *normale* Programmiererinnen und sind *more leightweight* als Modulsysteme oder Frameworks
- JavaScript Klassen und *klassische* Vererbung werden bewusst aussen vor gelassen, da sie
  - sicherlich viele Vorteile haben
  - nach wie vor umstritten sind
  - letztendlich auch auf die prototypische Struktur von JS abgebildet werden

  Außerdem ist die `klassische` Vererbung weithin bekannt

Outline

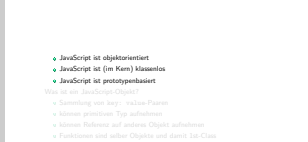
JavaScript: Wichtige Grundlagen

JavaScript als objektorientierte Sprache

- JavaScript ist objektorientiert
- JavaScript ist (im Kern) klassenlos
- JavaScript ist prototypenbasiert

Was ist ein JavaScript-Objekt?

- Sammlung von `key: value`-Paaren
- können primitiven Typ aufnehmen
- können Referenz auf anderes Objekt aufnehmen
- Funktionen sind selber Objekte und damit 1st-Class



JavaScript ist zwar weit verbreitet, jedoch gibt es immer wieder Missverständnisse bezüglich der Details und Wirkungsweisen
Viele Konfusionen entspringen daraus, dass JS als eine der wenigen OO-Sprachen `nicht` klassenbasiert ist, sondern *prototypenbasiert*
Um diese Missverständnisse auszuräumen soll das Objektsystem von JS erklärt werden:

- Objekterzeugung
- Objekte und Prototypen
- `this`-Binding
- dynamische Objekte

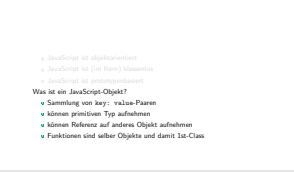
JavaScript: Wichtige Grundlagen

JavaScript als objektorientierte Sprache

- JavaScript ist objektorientiert
- JavaScript ist (im Kern) klassenlos
- JavaScript ist prototypenbasiert

Was ist ein JavaScript-Objekt?

- Sammlung von `key: value`-Paaren
- können primitiven Typ aufnehmen
- können Referenz auf anderes Objekt aufnehmen
- Funktionen sind selber Objekte und damit 1st-Class



JavaScript ist zwar weit verbreitet, jedoch gibt es immer wieder Missverständnisse bezüglich der Details und Wirkungsweisen
Viele Konfusionen entspringen daraus, dass JS als eine der wenigen OO-Sprachen `nicht` klassenbasiert ist, sondern *prototypenbasiert*

Was ist ein JavaScript-Objekt?

- Sammlung von `key: value`-Paaren
- können primitiven Typ aufnehmen
- können Referenz auf anderes Objekt aufnehmen
- Funktionen sind selber Objekte und damit 1st-Class

Objekterzeugung

Wie entstehen Objekte in JavaScript?

Vier Möglichkeiten Objekte zu erzeugen:

- Objektliterale

```
1 var empty = {}
2
3 var bello = {
4     name: 'Bello',
5     bark: function () {
6         console.log(`${this.name} says: Wuff-Wuff`);
7     },
8 }
```

- Konstruktorfunktionen
- mittels `Object.create()`
- Factories

- Objektliterale
- Leeres Objekt kann per {} erzeugt werden
- Objekte können einfach durch Angabe der *Dictionary* Einträge spezifiziert werden
- Konstruktorfunktionen
- mittels `Object.create()`
- Factories

Objekterzeugung

Wie entstehen Objekte in JavaScript?

Vier Möglichkeiten Objekte zu erzeugen:

- Objektliterale
- Konstruktorfunktionen

```
1 var Dog = function (dogName) {
2     this.name = dogName;
3     this.bark = function () {
4         console.log(`${this.name} says: Wuff-Wuff`);
5     }
6 }
7
8 var bello = new Dog('Bello');
```

- mittels `Object.create()`
- Factories

- Objektliterale
- Konstruktorfunktionen
- Jede funktion kann als Konstruktorfunktion dienen
- Funktion mit `new` aufgerufen ist Konstruktor und erzeugt neues Objekt
- Wenn Funktion keine explizite Rückgabe hat, dann wird neue erzeugtes Objekt zurückgegeben,
  ansonsten wird neues Objekt weggeschmissen
- mittels `Object.create()`
- Factories

Objekterzeugung

Wie entstehen Objekte in JavaScript?

Vier Möglichkeiten Objekte zu erzeugen:

- Objektliterale
- Konstruktorfunktionen
- mittels `Object.create()`

```
1 var protoDog = {
2     name: 'Bello', //default value
3     bark: function () {
4         console.log(`${this.name} says: Wuff-Wuff`);
5     },
6 }
7
8 var defaultDog = Object.create(protoDog);
```

- Factories

- Objektliterale
- Konstruktorfunktionen
- mittels `Object.create()`
- `Object.create(proto)` erzeugt neues, leeres Objekt und setzt den Prototypen auf übergebenes Objekt `proto`
- Factories

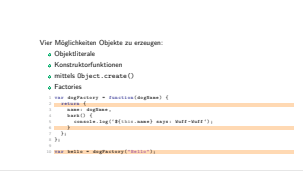
Objekterzeugung

Wie entstehen Objekte in JavaScript?

Vier Möglichkeiten Objekte zu erzeugen:

- Objektliterale
- Konstruktorfunktionen
- mittels `Object.create()`
- Factories

```
1  var dogFactory = function(dogName) {
2    return {
3      name: dogName,
4      bark() {
5        console.log(`${this.name} says: Wuff-Wuff`);
6      }
7    };
8  };
9
10 var bello = dogFactory("Bello");
```

- Objektliterale
- Konstruktorfunktionen
- mittels `Object.create()`
- Factories
- Jede Funktion kann ein irgendwie erzeugtes Objekt zurückgeben und damit als Factory dienen
- Im Beispiel wird ein Objektliteral zurückgegeben

Objekterzeugung

Wie entstehen Objekte in JavaScript?

Vier Möglichkeiten Objekte zu erzeugen:

- Objektliterale
- Konstruktorfunktionen
- mittels `Object.create()`
- Factories

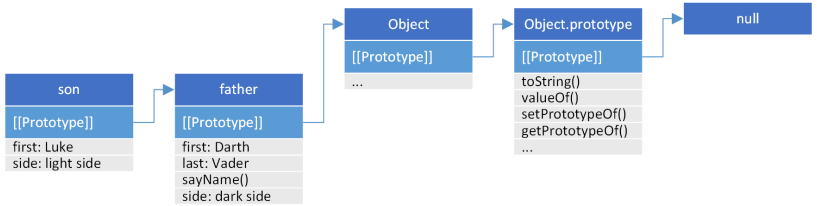
Vier Möglichkeiten Objekte zu erzeugen:
- Objektliterale
- Konstruktorfunktionen
- mittels `Object.create()`
- Factories

- Objektliterale
- Konstruktorfunktionen
- mittels `Object.create()`
- Factories
- Jede Funktion kann ein irgendwie erzeugtes Objekt zurückgeben und damit als Factory dienen
- Im Beispiel wird ein Objektliteral zurückgegeben

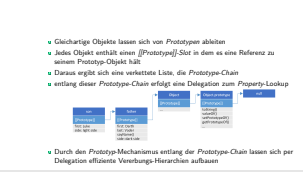
Prototypenbasierte Objektorientierung

Delegation entlang der Prototype-Chain

- Gleichartige Objekte lassen sich von *Prototypen* ableiten
- Jedes Objekt enthält einen *[[Prototype]]-Slot* in dem es eine Referenz zu seinem Prototyp-Objekt hält
- Daraus ergibt sich eine verkettete Liste, die *Prototype-Chain*
- entlang dieser *Prototype-Chain* erfolgt eine Delegation zum *Property*-Lookup



- Durch den *Prototyp*-Mechanismus entlang der *Prototype-Chain* lassen sich per Delegation effiziente Vererbungs-Hierarchien aufbauen



- *klassische* Objektorientierung basiert auf *Klassen*:
- In der Klassendefinition wird ein Bauplan für ein Objekt definiert
- Ein Objekt wird erzeugt, in dem entsprechend dieses Bauplans eine *materialisierte Objektkopie* erzeugt wird.
- ∄ Objekt ohne Klassendefinition

Javascript ist anders:

- JS kann Objekte *out of thin air* erzeugen
- Diese Objekte stehen für sich allein, haben aber eine Referenz auf ihren *Prototypen*
- Da jedes Objekt einen *[[Prototype]]-Slot* hat, entsteht eine einfach verkettete Liste von Objekten
- entlang dieser *Prototype-Chain* erfolgt eine Delegation zum *Property*-Lookup

prototypeChain.js

live Code Beispiel


```
let father = {
    first: "Darth",
    last: "Skywalker",

    sayName: function () {
        console.log(`My name is ${this.first} ${this.last}.`);
    }
}

let son = {};   //empty object with Prototype set to Object
Object.setPrototypeOf(son, father); //performance penalty!!! Better: let son = Object.create(father)
console.log(`son has own keys: [${Object.keys(son)}]`);   //son has own keys: []

son.first = "Luke"; //becomes an own property of son and will shadow father's prop of the same name
son.side = "light side"; //becomes an own property of son

son.sayName();      //My name is Luke Skywalker.
father.sayName();   //My name is Darth Skywalker.

father.last = "Vader"
father.side = "dark side"; //this becomes an own property of father


son.sayName();      //My name is Luke Vader.    //Oops, the last name changed on his Prototype Object
console.log(son.side);  //light side

father.sayName();   //My name is Darth Vader.
console.log(father.side);   //dark side

console.log(`Object.keys(son): [${Object.keys(son)}]`);   //Object.keys(son): [first,side]
console.log(`Object.keys(father): [${Object.keys(father)}]`); //Object.keys(father): [first,last,sayName,side]
```


Code Beispiel: `prototypeChain.js`

- `son` Objekt hat zunächst keine eigenen Props
- `son` bekommt eigene Props
- `son.first` wird als eigenen Prop erstellt und überdeckt `father.first`
- Zuweisung zu `father.last` beeinflusst auch Ausgaben von `son.last`, da keine eigenen Prop
- Keys der beiden Objekte wie im Bild dargestellt
- Bei der Zuweisungen auf gemeinsame Properties müssen die Regeln des Shadowing beachtet werden:
  - Wert-Properties auf dem Prototypen werden bei Zuweisung verdeckt
  - Objektreferenzen auf dem Prototypen werden tatsächlich gemeinsam genutzt

this-Binding

Das Schlüsselwort `this` verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.

- *new binding* – Aufruf einer Konstruktorfunktion mit `new`
  - Ein neues leeres Objekt wird erzeugt
  - Der Prototyp-Link wird auf `constrFn.prototype` gesetzt
  - `this` wird an das neue Objekt gebunden
  - Die Konstruktorfunktion wird ausgeführt
  - Das neue Objekt wird implizit zurückgegeben
- *explicit binding* – `Function.prototype.call()`
- *implicit binding* – `obj.method()`
- *default binding*
- *lexical binding* – Arrow-Funktionen in ES6

Das Schlüsselwort this verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.
new binding – Aufruf einer Konstruktorfunktion mit new
Ein neues leeres Objekt wird erzeugt
Der Prototyp-Link wird auf constrFn.prototype gesetzt
this wird an das neue Objekt gebunden
Die Konstruktorfunktion wird ausgeführt
Das neue Objekt wird implizit zurückgegeben
explicit binding – Function.prototype.call()
implicit binding – obj.method()
default binding
lexical binding – Arrow-Funktionen in ES6

new binding – Aufruf einer Konstruktorfunktion mit `new`

- Ein neues leeres Objekt wird erzeugt
- Der Prototyp-Link wird auf `consFn.prototype` gesetzt
- `this` wird an das neue Objekt gebunden
- Die Konstruktorfunktion wird ausgeführt
- Das neue Objekt wird implizit zurückgegeben

this-Binding

Das Schlüsselwort `this` verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.

- *new binding* – Aufruf einer Konstruktorfunktion mit `new`
- *explicit binding* – `Function.prototype.call()`
  - Das `this`-Binding kann explizit gesetzt werden
  - Dazu wird eine Methode aufgerufen über `method.call(obj, ...args)`
  - `this` wird an das übergebene Objekt `obj` gebunden
- *implicit binding* – `obj.method()`
- *default binding*
- *lexical binding* – Arrow-Funktionen in ES6

Das Schlüsselwort this verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.
new binding – Aufruf einer Konstruktorfunktion mit new
explicit binding – Function.prototype.call()
Das this-Binding kann explizit gesetzt werden
Dazu wird eine Methode aufgerufen über method.call(obj, ...args)
this wird an das übergebene Objekt obj gebunden
implicit binding – obj.method()
default binding
lexical binding – Arrow-Funktionen in ES6

explicit binding – `Function.prototype.call()`

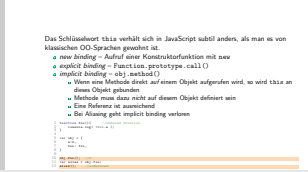
- Das `this`-Binding kann explizit gesetzt werden
- Dazu wird eine Methode aufgerufen über `method.call(obj, ...args)`
- `this` wird an das übergebene Objekt `obj` gebunden

this-Binding

Das Schlüsselwort `this` verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.

- *new binding* – Aufruf einer Konstruktorfunktion mit `new`
- *explicit binding* – `Function.prototype.call()`
- *implicit binding* – `obj.method()`
  - Wenn eine Methode direkt *auf* einem Objekt aufgerufen wird, so wird `this` an dieses Objekt gebunden
  - Methode muss dazu *nicht* auf diesem Objekt definiert sein
  - Eine Referenz ist ausreichend
  - Bei Aliasing geht implicit binding verloren

```
1  function foo(){     //unbound function
2      console.log( this.a );
3  }
4
5  var obj = {
6      a:2,
7      foo: foo,
8  }
9
10 obj.foo();  //2
11 var alias = obj.foo;
12 alias();    //undefined
```

implicit binding – `obj.method()`

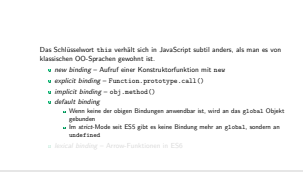
- Wenn eine Methode direkt *auf* einem Objekt aufgerufen wird, so wird `this` an dieses Objekt gebunden
- Methode muss dazu *nicht* auf diesem Objekt definiert sein
- Eine Referenz ist ausreichend
- Bei Aliasing geht implicit binding verloren

```
1  function foo(){     //unbound function
2      console.log( this.a );
3  }
4
5  var obj = {
6      a:2,
7      foo: foo,
8  }
9
10 obj.foo();  //2
11 var alias = obj.foo;
12 alias();    //undefined
```

this-Binding

Das Schlüsselwort `this` verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.

- *new binding* – Aufruf einer Konstruktorfunktion mit `new`
- *explicit binding* – `Function.prototype.call()`
- *implicit binding* – `obj.method()`
- *default binding*
  - Wenn keine der obigen Bindungen anwendbar ist, wird an das `global` Objekt gebunden
  - Im *strict*-Mode seit ES5 gibt es keine Bindung mehr an `global`, sondern an `undefined`
- *lexical binding* – Arrow-Funktionen in ES6



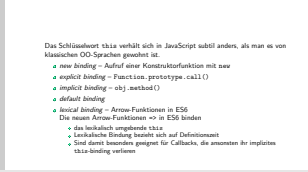
default binding

- Wenn keine der obigen Bindungen anwendbar ist, wird an das `global` Objekt gebunden
- Im *strict*-Mode seit ES5 gibt es keine Bindung mehr an `global`, sondern an `undefined`

this-Binding

Das Schlüsselwort `this` verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.

- *new binding* – Aufruf einer Konstruktorfunktion mit `new`
- *explicit binding* – `Function.prototype.call()`
- *implicit binding* – `obj.method()`
- *default binding*
- *lexical binding* – Arrow-Funktionen in ES6
  Die neuen Arrow-Funktionen => in ES6 binden
  - das lexikalisch umgebende `this`
  - Lexikalische Bindung bezieht sich auf Definitionszeit
  - Sind damit besonders geeignet für Callbacks, die ansonsten ihr implizites `this`-binding verlieren



lexical binding – Arrow-Funktionen in ES6
Die neuen Arrow-Funktionen => in ES6 binden

- das lexikalisch umgebende `this`
- Lexikalische Bindung bezieht sich auf Definitionszeit
- Sind damit besonders geeignet für Callbacks, die ansonsten ihr implizites `this`-binding verlieren

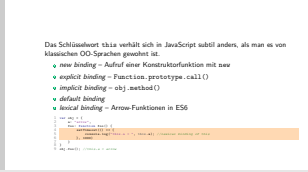
```
1 var obj = {
2     a: "arrow",
3     foo: function foo() {
4         setTimeout(() => {
5             console.log("this.a = ", this.a); //lexical binding of this
6         }, 1000)
7     }
8 }
9 obj.foo(); //this.a = arrow
```

this-Binding

Das Schlüsselwort `this` verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.

- *new binding* – Aufruf einer Konstruktorfunktion mit `new`
- *explicit binding* – `Function.prototype.call()`
- *implicit binding* – `obj.method()`
- *default binding*
- *lexical binding* – Arrow-Funktionen in ES6

```
1 var obj = {
2     a: "arrow",
3     foo: function foo() {
4         setTimeout(() => {
5             console.log("this.a = ", this.a); //lexical binding of this
6         }, 1000)
7     }
8 }
9 obj.foo(); //this.a = arrow
```

lexical binding – Arrow-Funktionen in ES6
Die neuen Arrow-Funktionen => in ES6 binden

- das lexikalisch umgebende `this`
- Lexikalische Bindung bezieht sich auf Definitionszeit
- Sind damit besonders geeignet für Callbacks, die ansonsten ihr implizites `this`-binding verlieren


```
1 var obj = {
2     a: "arrow",
3     foo: function foo() {
4         setTimeout(() => {
5             console.log("this.a = ", this.a); //lexical binding of this
6         }, 1000)
7     }
8 }
9 obj.foo(); //this.a = arrow
```

this-Binding

Das Schlüsselwort `this` verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.

- *new binding* – Aufruf einer Konstruktorfunktion mit `new`
- *explicit binding* – `Function.prototype.call()`
- *implicit binding* – `obj.method()`
- *default binding*
- *lexical binding* – Arrow-Funktionen in ES6

Das Schlüsselwort this verhält sich in JavaScript subtil anders, als man es von klassischen OO-Sprachen gewohnt ist.
- new binding – Aufruf einer Konstruktorfunktion mit new
- explicit binding – Function.prototype.call()
- implicit binding – obj.method()
- default binding
- lexical binding – Arrow-Funktionen in ES6

lexical binding – Arrow-Funktionen in ES6
Die neuen Arrow-Funktionen => in ES6 binden

- das lexikalisch umgebende `this`
- Lexikalische Bindung bezieht sich auf Definitionszeit
- Sind damit besonders geeignet für Callbacks, die ansonsten ihr implizites `this`-binding verlieren

```
1 var obj = {
2     a: "arrow",
3     foo: function foo() {
4         setTimeout(() => {
5             console.log("this.a = ", this.a); //lexical binding of this
6         }, 1000)
7     }
8 }
9 obj.foo(); //this.a = arrow
```

Dynamische Objekte

Ein Objekt in JavaScript ist dynamisch:
Zur Laufzeit können

- Properties hinzugefügt werden
- Properties gelöscht werden
- Referenzen geändert werden

JavaScript Objekte sind jederzeit veränderbar
Damit ist es schwierig den Typ eines Objekts zu bestimmen
→ Duck-Typing:

> *if it looks like a duck, and it quacks like a duck,*
> *it must be a duck [Simpson, 2014, p. 141]*

Ein Objekt in JavaScript ist dynamisch:
Zur Laufzeit können
- Properties hinzugefügt werden
- Properties gelöscht werden
- Referenzen geändert werden

JavaScript Objekte sind jederzeit veränderbar
Damit ist es schwierig den Typ eines Objekts zu bestimmen
→ Duck-Typing:
if it looks like a duck, and it quacks like a duck,
it must be a duck [Simpson, 2014, p. 141]

Ein Objekt in JavaScript ist dynamisch:
Zur Laufzeit können

- Properties hinzugefügt werden
- Properties gelöscht werden
- Referenzen geändert werden

JavaScript Objekte sind jederzeit veränderbar
Damit ist es schwierig den Typ eines Objekts zu bestimmen

→ Duck-Typing:

if it looks like a duck, and it quacks like a duck,
it must be a duck [Simpson, 2014, p. 141]

Outline

Method Borrowing

Wiederverwendung einer einzelnen Methode

Eine Methode eines Objekts kann explizit an ein anderes Objekt gebunden werden und darauf angewendet werden.

Dazu gibt es die Funktionen `call()` und `apply()`

```
1 // call() example
2 notmyobj.doStuff.call(myobj, param1, p2, p3);
3 // apply() example
4 notmyobj.doStuff.apply(myobj, [param1, p2, p3]);
```


Eine Methode eines Objekts kann explizit an ein anderes Objekt gebunden werden und darauf angewendet werden.

Dazu gibt es die Funktionen call() und apply()

```
1 // call() example
2 notmyobj.doStuff.call(myobj, param1, p2, p3);
3 // apply() example
4 notmyobj.doStuff.apply(myobj, [param1, p2, p3]);
```

- Das ist der eben besprochene Fall des explicit `this`-Binding
- Dazu gibt es die Funktionen `call()` und `apply()` definiert auf `Function.prototype` und per Delegation auf Funktionen verfügbar.
- Unterscheiden sich nur in der Art der Parameterübergabe
-
- Beispiel:

Method Borrowing

Wiederverwendung einer einzelnen Methode

Eine Methode eines Objekts kann explizit an ein anderes Objekt gebunden werden und darauf angewendet werden.

Dazu gibt es die Funktionen `call()` und `apply()`

```
1 var toArray = function () {
2     return Array.prototype.slice.call(arguments);
3 }
4
5 console.log(toArray(1, 4, 3, 2)); //[ 1, 4, 3, 2 ]
```

Die Argumente eines Funktionsaufrufs sind in der Variable `arguments` verfügbar. Das ist kein Array, sondern nur ein *Array-like*-Objekt (d. h. es besitzt einen Iterator) Per Method-Borrowing kann die Methode `Array.prototype.slice` darauf angewendet werden und so ein *echtes* Array liefern.

Eine Methode eines Objekts kann explizit an ein anderes Objekt gebunden werden und darauf angewendet werden.

Dazu gibt es die Funktionen call() und apply()

```
var toArray = function () {
    return Array.prototype.slice.call(arguments);
}

console.log(toArray(1, 4, 3, 2)); //[ 1, 4, 3, 2 ]
```

Die Argumente eines Funktionsaufrufs sind in der Variable arguments verfügbar. Das ist kein Array, sondern nur ein Array-like-Objekt (d. h. es besitzt einen Iterator) Per Method-Borrowing kann die Methode Array.prototype.slice darauf angewendet werden und so ein echtes Array liefern.

- Das ist der eben besprochene Fall des explicit `this`-Binding
- Dazu gibt es die Funktionen `call()` und `apply()` definiert auf `Function.prototype` und per Delegation auf Funktionen verfügbar.
- Unterscheiden sich nur in der Art der Parameterübergabe
-
- Beispiel:

Delegation als Vererbungsmechanismus

Delegation entlang der Prototype-Chain

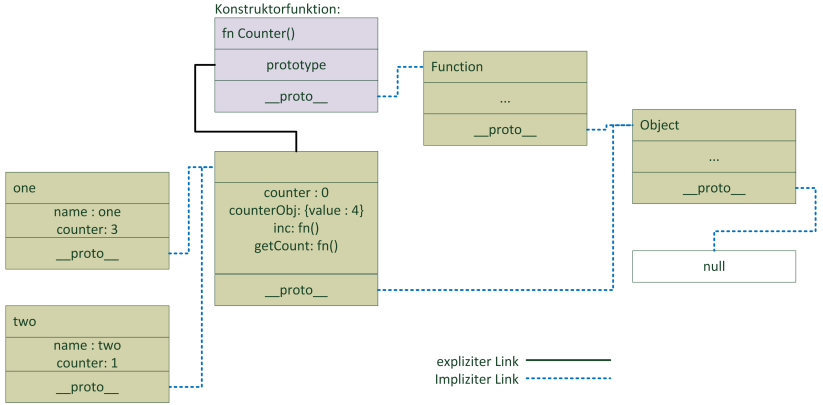
- Objekte bauen entlang ihrer Prototype-Chain aufeinander auf
- Delegation ermöglicht klassenähnliche Vererbung in JavaScript
- Gemeinsame Properties und Methoden werden auf dem Prototypen definiert
- Bei der Zuweisungen auf gemeinsame Properties müssen die Regeln des Shadowing beachtet werden:
  - Wert-Properties auf dem Prototypen werden bei Zuweisung verdeckt
  - Objektreferenzen auf dem Prototypen werden tatsächlich gemeinsam genutzt

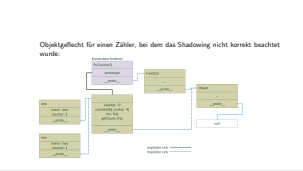
- Objekte bauen entlang ihrer Prototype-Chain aufeinander auf
- Delegation ermöglicht klassenähnliche Vererbung in JavaScript
- Gemeinsame Properties und Methoden werden auf dem Prototypen definiert
- Bei der Zuweisungen auf gemeinsame Properties müssen die Regeln des Shadowing beachtet werden:
  - Wert-Properties auf dem Prototypen werden bei Zuweisung verdeckt
  - Objektreferenzen auf dem Prototypen werden tatsächlich gemeinsam genutzt

- Dies geht bei Objekterstellung über Konstruktorfunktionen besonders leicht durch Definition auf `ConstructorFn.prototype`
- Auch bei Erzeugung mittels `Object.create()` lässt sich der Prototyp direkt angeben
-
- Shadowing bezeichnet as Anlegen einer neuen Property bei schreibendem Zugriff
- Wenn unten in der Chain eine Property geschrieben wird, so wird dort eine neue Property angelegt
- Diese neue Property verdeckt die weiter oben in der Kette leigende Property des Prototypen

Vererbung
Beispiel mit Shadowing

Objektgeflecht für einen Zähler, bei dem das Shadowing nicht korrekt beachtet wurde:





Code-Beispiel: `../codesnips/protoCounterConstructor.js`

- `.prototype` ist *nicht* zu Verwechseln mit `__proto__` der Konstruktorfunktion
- `Counter.counter` ist ein primitiver Wert. Daher wird der Wert verborgen (Shadowing)
- `Counter.counterObj` ist eine Referenz auf ein weiteres (nicht dargestelltes) Objekt
- Daher greifen alle abgeleiteten Objekte auf das gleiche Objekt zu und es erfolgt *kein Shadowing*

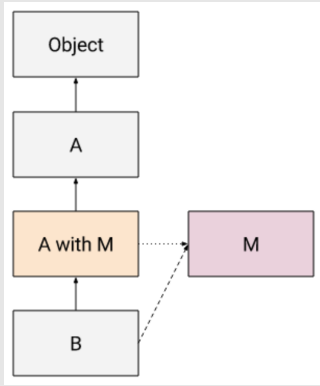
Grenzen der Delegation

Nicht alles ist eine Baumstruktur

- Sprachen ohne Mehrfachvererbung bieten nur Baumstrukturen zur Vererbung
- Baumstruktur ist nicht immer geeignet, um die Objekteigenschaften abzubilden
- Schwierig damit *orthogonale* Eigenschaften abzubilden

Sprachen ohne Mehrfachvererbung bieten nur Baumstrukturen zur Vererbung
Baumstruktur ist nicht immer geeignet, um die Objekteigenschaften abzubilden
Schwierig damit orthogonale Eigenschaften abzubilden

- *Mixins* zur *orthogonalen* Objekterweiterung
- in klassischen Sprachen eine *abstrakte Subklasse*
- Mixin-Klasse stützt sich auf Eigenschaften der Zielklasse



Grenzen der Delegation

Nicht alles ist eine Baumstruktur

- Sprachen ohne Mehrfachvererbung bieten nur Baumstrukturen zur Vererbung
- Baumstruktur ist nicht immer geeignet, um die Objekteigenschaften abzubilden
- Schwierig damit *orthogonale* Eigenschaften abzubilden
- *Mixins* zur orthogonalen Objekterweiterung

Source: Werbung aus https:

- Klassische Sprachen bieten in der Regel nur *abstrakte Superklassen*
- abstrakte *Sub*klassen müssen unterstützt werden um Mixins anzuwenden
- Objekte lassen sich nicht ohne Weiteres erweitern

- In JavaScript sehr einfach, da Objekte dynamisch sind
- Properties eines Mixin-Objekts können auf ein Zielobjekt kopiert werden

Grenzen der Delegation

Nicht alles ist eine Baumstruktur

- Sprachen ohne Mehrfachvererbung bieten nur Baumstrukturen zur Vererbung
- Baumstruktur ist nicht immer geeignet, um die Objekteigenschaften abzubilden
- Schwierig damit *orthogonale* Eigenschaften abzubilden
- *Mixins* zur orthogonalen Objekterweiterung

- In JavaScript sehr einfach, da Objekte dynamisch sind
- Properties eines Mixin-Objekts können auf ein Zielobjekt kopiert werden

- Sprachen ohne Mehrfachvererbung bieten nur Baumstrukturen zur Vererbung
- Baumstruktur ist nicht immer geeignet, um die Objekteigenschaften abzubilden
- Schwierig damit orthogonale Eigenschaften abzubilden
- *Mixins* zur orthogonalen Objekterweiterung

- In JavaScript sehr einfach, da Objekte dynamisch sind
- Properties eines Mixin-Objekts können auf ein Zielobjekt kopiert werden

- Klassische Sprachen bieten in der Regel nur *abstrakte Superklassen*
- abstrakte *Sub*klassen müssen unterstützt werden um Mixins anzuwenden
- Objekte lassen sich nicht ohne Weiteres erweitern

- In JavaScript sehr einfach, da Objekte dynamisch sind
- Properties eines Mixin-Objekts können auf ein Zielobjekt kopiert werden

Wiederverwendung durch Kopieren

JavaScript-Objekte sind dynamisch

- dynamischen Objekten können zur Laufzeit Properties hinzugefügt werden
- Code-Wiederverwendung durch *Kopieren* ist möglich
- `Object.assign(target, ...sources)` kopiert *eigene* Properties aus `sources` in das `target` Objekt
- Code-Wiederverwendung ohne Prototype-Referenz
- Bereits vorhandenen Properties gleichen Namens werden überschrieben

- dynamischen Objekten können zur Laufzeit Properties hinzugefügt werden
- Code-Wiederverwendung durch *Kopieren* ist möglich
- `Object.assign(target, ...sources)` kopiert *eigene* Properties aus `sources` in das `target` Objekt
- Code-Wiederverwendung ohne Prototype-Referenz
- Bereits vorhandenen Properties gleichen Namens werden überschrieben

Es können meherere `source` Objekte angegeben werden

- Es wird lediglich eine flache Kopie erzeugt: d. h. Objektreferenzen werden nicht dupliziert
- Flache Kopie
- Gleiches Problem wie bei Prototype-Chain und Objektreferenzen
-
- es lässt sich relativ leicht eine *extendDeep*-Funktion bauen
- eine tiefe Kopie kann eine sehr teure Operation sein
- leiber bleiben lassen

Last wins bei Namensgleichheit
==> Es kommt auf Reihenfolge der Argumente an

Mixins-Anwendung

Code Beispiel


```
1  var mixDeveloper = {languages: [], patterns: [], /*...*/}
2  var mixEmployee = {personnelNumber: 0, /*...*/}
3  var mixFreelancer = {hourlyRate: 100, /*...*/}
4  var protoPerson = {initPerson(name) {/*...*/}, /*...*/}
5
6
7  let felix = Object.create(protoPerson).initPerson('Felix');
8  Object.assign(felix, mixDeveloper, mixFreelancer);
9
10 felix.initDeveloper(['JS', 'C'], ['Mixins', 'Decorators'])
11     .initFreelance(120);
12
13
14 let john = Object.create(protoPerson).initPerson('John');
15 Object.assign(john, mixDeveloper, mixEmployee);
16
17 john.initDeveloper(['Java'], ['Singletons', 'Facade'])
18     .initEmployee(666, 30000);
```


- Codebeispiel `../codesnips/simpleMixin.js`
-
- Zunächst wird ein Objekt mit Prototyp `protoPerson` angelegt
-
- Dann wird die `Developer` bzw. die `Freelancer` oder `Employee` Eigenschaft eingemixt

Functional Mixins

Eine andere Sichtweise

- Mixins sind ein Amalgam aus mehreren Objekten
- Mixin-Objekte sind abstrakt und für sich allein nutzlos
- Kopierfunktion notwendig
- Aufwändig, wenn weitere Parameter notwendig sind
- Information Hiding bei einfacher Kopie schwierig

Neue Sichtweise:
Mixin als Funktion, die ein übergebenes Objekt erweitert

- Entspricht dem *Decorator*-Pattern:
  Objekte werden durch Anwenden einer Funktion erweitert
- Es ist ein Funktionsaufruf, bei dem leicht Parameter übergeben werden können
- Function Closure zur Datenkapselung

- Mixins sind ein Amalgam aus mehreren Objekten
- Mixin-Objekte sind abstrakt und für sich allein nutzlos
- Kopierfunktion notwendig
- Aufwändig, wenn weitere Parameter notwendig sind
- Information Hiding bei einfacher Kopie schwierig

Functional Mixins

Eine andere Sichtweise

- Mixins sind ein Amalgam aus mehreren Objekten
- Mixin-Objekte sind abstrakt und für sich allein nutzlos
- Kopierfunktion notwendig
- Aufwändig, wenn weitere Parameter notwendig sind
- Information Hiding bei einfacher Kopie schwierig

Neue Sichtweise:
Mixin als Funktion, die ein übergebenes Objekt erweitert

- Entspricht dem *Decorator*-Pattern:
  Objekte werden durch Anwenden einer Funktion erweitert
- Es ist ein Funktionsaufruf, bei dem leicht Parameter übergeben werden können
- Function Closure zur Datenkapselung

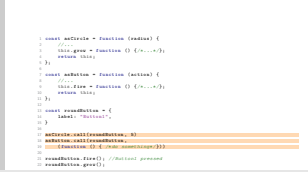
Neue Sichtweise:
Mixin als Funktion, die ein übergebenes Objekt erweitert

- Entspricht dem *Decorator*-Pattern:
  Objekte werden durch Anwenden einer Funktion erweitert
- Es ist ein Funktionsaufruf, bei dem leicht Parameter übergeben werden können
- Function Closure zur Datenkapselung

Functional Mixins

Ein einfaches Beispiel


```
1  const asCircle = function (radius) {
2      //...
3      this.grow = function () {/*...*/};
4      return this;
5  };
6
7  const asButton = function (action) {
8      //...
9      this.fire = function () {/*...*/};
10     return this;
11 };
12
13 const roundButton = {
14     label: "Button1",
15 }
16
17 asCircle.call(roundButton, 5)
18 asButton.call(roundButton,
19     (function () { /*do something*/}))
20
21 roundButton.fire(); //Button1 pressed
22 roundButton.grow();
```

Live Code-Beispiel: ../codesnips/simpleFunctionalMixin.js

- Das einfache Objekt `roundButton` wird mit weiterer Funktionalität versehen
- Beim Dekorieren des ursprünglichen Objekt können leicht Parameter übergeben werden (einfacher Funktionsaufruf)
- Es ergibt sich eine ausdrucksstarke Syntax
- Functional Mixins sind selber für die Anreicherung der Objekte zuständig
  - Sind nicht von einer externen Kopierfunktion `Object.assign()` abhängig
  - können selber entscheiden, wie sie mit Namenskonflikten umgehen
  - können *super-calls* nachbilden, wenn eine gleichnamige Funktionalität schon vorhanden ist
  - Da Funktionen eine Closure bilden können Mixin-spezifische Properties sehr leicht gekapselt werden

Functional Mixins in einer Factory

Moderne ES6-Syntax ermöglicht Ausdrucksstärke

JavaScript ist funktional:

- Funktionen sind *composable*, d. h. sie lassen sich aufeinander Higher-Order-Function anwenden
- `pipe(...fns)`-Funktion wendet alle übergebenen Funktionen auf ein Objekt an

Factories mit mehreren *functional Mixins* ergeben sehr ausdrucksstarke Programme:

```
1  const withFlying = o => {
2      return {...o, fly() {/*...*/}, land() {/*...*/},isFlying: () => /*...*/}}
3
4  const withBattery = ({ capacity }) => o => {
5      return {...o, draw(percent) {/*...*/}, getCapacity() {/*...*/}}}
6
7  const createDrone = ({ capacity = '3000mAh' }) => pipe(
8      withFlying,
9      withBattery({ capacity })
10 )({});
11
12 const myDrone = createDrone();  //Drone with default battery
13
14 const myDrone1 = createDrone({ capacity: '666mAh' });   //smaller battery
```


JavaScript ist funktional:

- Funktionen sind *composable*, d. h. sie lassen sich aufeinander Higher-Order-Function anwenden
- `pipe(...fns)`-Funktion wendet alle übergebenen Funktionen auf ein Objekt an

Factories mit mehreren *functional Mixins* ergeben sehr ausdrucksstarke Programme:

Functional Mixins in einer Factory

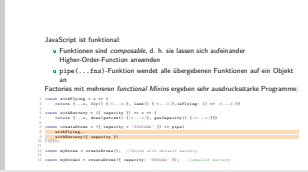
Moderne ES6-Syntax ermöglicht Ausdrucksstärke

JavaScript ist funktional:

- Funktionen sind *composable*, d. h. sie lassen sich aufeinander Higher-Order-Function anwenden
- `pipe(...fns)`-Funktion wendet alle übergebenen Funktionen auf ein Objekt an

Factories mit mehreren *functional Mixins* ergeben sehr ausdrucksstarke Programme:

```
1  const withFlying = o => {
2      return {...o, fly() {/*...*/}, land() {/*...*/},isFlying: () => /*...*/}}
3
4  const withBattery = ({ capacity }) => o => {
5      return {...o, draw(percent) {/*...*/}, getCapacity() {/*...*/}}}
6
7  const createDrone = ({ capacity = '3000mAh' }) => pipe(
8      withFlying,
9      withBattery({ capacity })
10 )({});
11
12 const myDrone = createDrone();  //Drone with default battery
13
14 const myDrone1 = createDrone({ capacity: '666mAh' });   //smaller battery
```

```
1  const withFlying = o => {
2      return {...o, fly() {/*...*/}, land() {/*...*/},isFlying: () => /*...*/}}
3
4  const withBattery = ({ capacity }) => o => {
5      return {...o, draw(percent) {/*...*/}, getCapacity() {/*...*/}}}
6
7  const createDrone = ({ capacity = '3000mAh' }) => pipe(
8      withFlying,
9      withBattery({ capacity })
10 )({});
11
12 const myDrone = createDrone();  //Drone with default battery
13
14 const myDrone1 = createDrone({ capacity: '666mAh' });   //smaller battery
```

Live Code-Beispiel: ../codesnips/dronesFactory.js

Outline

Einführung
JavaScript: Wichtige Grundlagen
Objekterzeugung
Prototype-Chain
this-Binding
dynamische Objekte
Code-Reuse
Method Borrowing
Delegation und Vererbung
Mixins für orthogonalen Code-Resue
Functional Mixins
Fazit
Literatur

2019-03-27
Borrow – Inherit – Mix
Fazit
Outline

Fazit

Kritik und Ausblick

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:

- Method Borrowing
  - sehr enge Kopplung
  - Implementierungsdetails des Wirts-Objekts sind kritisch
  - Wenn möglich Hilfsfunktionen lieber in Modulen implementieren und Objektreferenzen explizit übergeben
- Inheritance
- Mixins
- Functional Mixins

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:
Method Borrowing

- sehr enge Kopplung
- Implementierungsdetails des Wirts-Objekts sind kritisch
- Wenn möglich Hilfsfunktionen lieber in Modulen implementieren und Objektreferenzen explizit übergeben

Fazit

Kritik und Ausblick

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:

- Method Borrowing – einfache Anwendung, enge Kopplung
- Inheritance
  - sehr enge Kopplung
  - starre Baum-Hierarchie
- Mixins
- Functional Mixins

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:
Inheritance

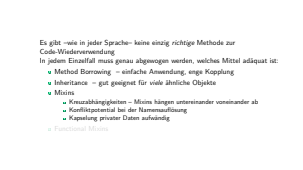
- sehr enge Kopplung
- starre Baum-Hierarchie

Fazit

Kritik und Ausblick

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:

- Method Borrowing – einfache Anwendung, enge Kopplung
- Inheritance – gut geeignet für *viele* ähnliche Objekte
- Mixins
  - Kreuzabhängigkeiten – Mixins hängen untereinander voneinander ab
  - Konfliktpotential bei der Namensauflösung
  - Kapselung privater Daten aufwändig
- Functional Mixins



Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:
Mixins

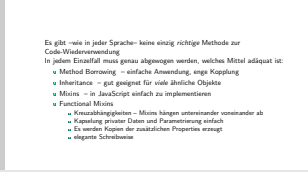
- Kreuzabhängigkeiten – Mixins hängen untereinander voneinander ab
- Konfliktpotential bei der Namensauflösung
- Kapselung privater Daten aufwändig

Fazit

Kritik und Ausblick

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:

- Method Borrowing – einfache Anwendung, enge Kopplung
- Inheritance – gut geeignet für *viele* ähnliche Objekte
- Mixins – in JavaScript einfach zu implementieren
- Functional Mixins
  - Kreuzabhängigkeiten – Mixins hängen untereinander voneinander ab
  - Kapselung privater Daten und Parametrierung einfach
  - Es werden Kopien der zusätzlichen Properties erzeugt
  - elegante Schreibweise



Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:
Functional Mixins

- Kreuzabhängigkeiten – Mixins hängen untereinander voneinander ab
- Kapselung privater Daten und Parametrierung einfach
- Es werden Kopien der zusätzlichen Properties erzeugt
- elegante Schreibweise

Fazit

Kritik und Ausblick

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:

- Method Borrowing – einfache Anwendung, enge Kopplung
- Inheritance – gut geeignet für *viele* ähnliche Objekte
- Mixins – in JavaScript einfach zu implementieren
- Functional Mixins – sehr ausdrucksstarke Schreibweise

Es gibt –wie in jeder Sprache– keine einzig *richtige* Methode zur Code-Wiederverwendung
In jedem Einzelfall muss genau abgewogen werden, welches Mittel adäquat ist:

Es geht nicht ohne . . .


```
1 const createDrone = ({ capacity = '3000mAh', rotors=4, pet='' }) => pipe(
2     withBattery({ capacity }),
3     withFlying,
4     withRotors({ rotors }),
5     withRemote,
6     withBelovedPet({ pet }),
7 )({});
8
9 const catCopter = createDrone({ pet = 'cat' });
```

Source: Cat-Copter aus https://metro.co.uk/2015/10/01/

1 const createDrone = ({ capacity = '3000mAh', rotors=4, pet='' }) => pipe(
2 withBattery({ capacity }),
3 withFlying,
4 withRotors({ rotors }),
5 withRemote,
6 withBelovedPet({ pet }),
7)({});
8
9 const catCopter = createDrone({ pet = 'cat' });

- Katzenfotos müssen in jedem Javascript Vortrag sein
-
- Also bauen wir uns ein Katze

Es geht nicht ohne . . .

. . . Katzenfotos


```
1 const createDrone = ({ capacity = '3000mAh', rotors=4, pet='' }) => pipe(
2     withBattery({ capacity }),
3     withFlying,
4     withRotors({ rotors }),
5     withRemote,
6     withBelovedPet({ pet }),
7 )({});
8
9 const catCopter = createDrone({ pet = 'cat' });
```

Source: Cat-Copter aus https://metro.co.uk/2015/10/01/

- Katzenfotos müssen in jedem Javascript Vortrag sein
-
- Also bauen wir uns ein Katze

Literatur

[Croll 2011] Croll, Angus: *A Fresh Look at JavaScript Mixins*. Mai 2011. – URL https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/. – Zugriffsdatum: 2018-11-25

[Elliott 2014] Elliott, Eric: *Programming JavaScript Applications*. First edition. Beijing; Sebastopol : O'Reilly, 2014. – OCLC: ocn867765966. – ISBN 978-1-4919-5029-6

[Elliott 2017] Elliott, Eric: *Functional Mixins*. Juni 2017. – URL https://medium.com/javascript-scene/functional-mixins-composing-software-ffb66d5e731c. – Zugriffsdatum: 2018-12-18

[Simpson 2014] Simpson, Kyle: *This & Object Prototypes*. First edition. Beijing; Sebastopol, CA : O'Reilly, 2014 (You don't know JS). – OCLC: ocn891619771. – ISBN 978-1-4919-0415-2

[Stefanov 2010] Stefanov, Stoyan: *JavaScript Patterns : [Build Better Applications with Coding and Design Patterns]*. 1. ed. Beijing [u.a.] : O'Reilly, 2010 (Yahoo! Press). – ISBN 978-0-596-80675-0

[TC39 und Terlson 2018] TC39, ECMA ; Terlson, Brian: *ECMAScript 2018 Language Specification*. 9th. Geneva : ECMA International, 2018. – URL https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. – Zugriffsdatum: 2018-11-28