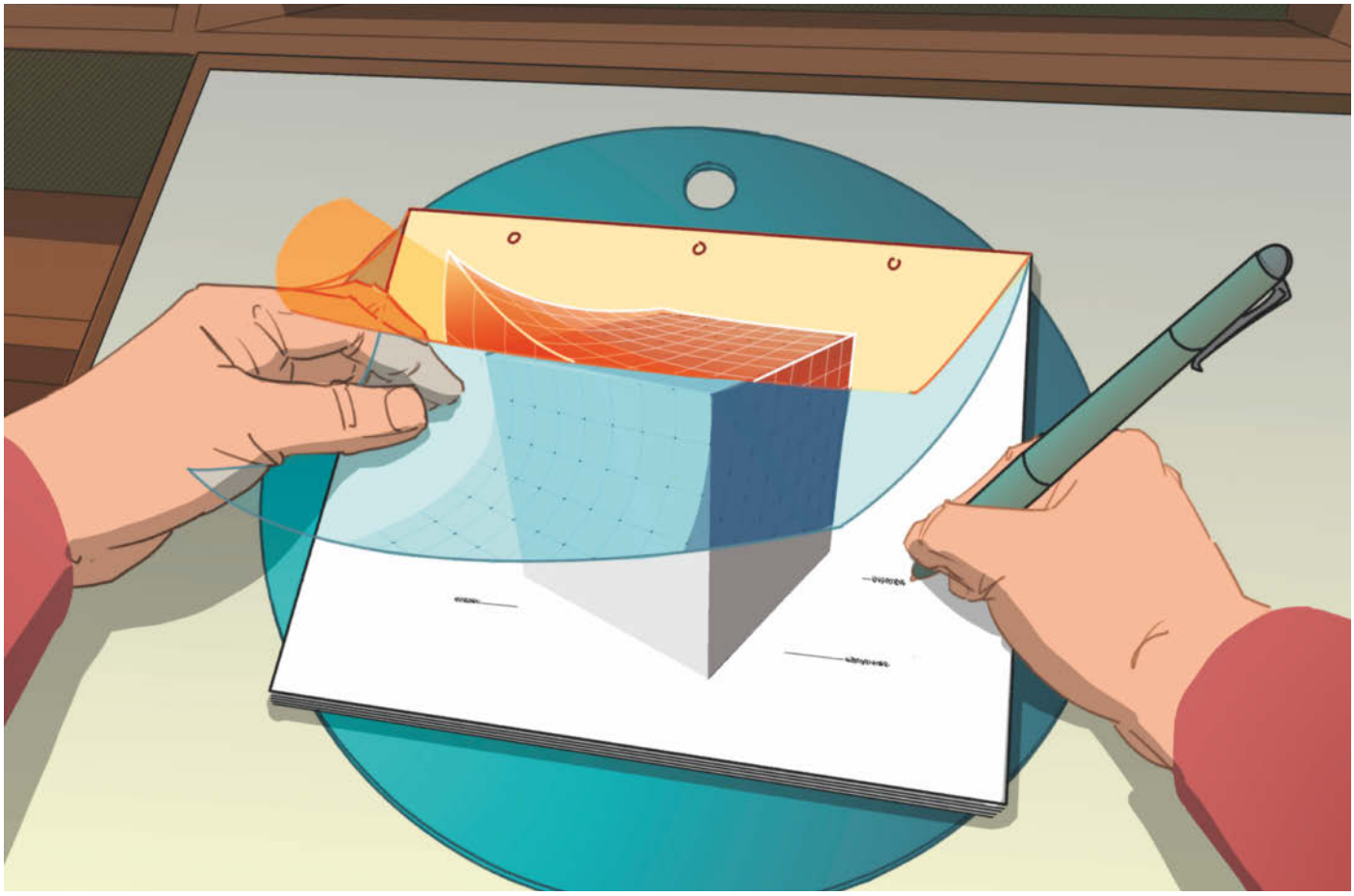


Bild: Thorsten Hübner

Blaupausen

Wie Objekte in JavaScript funktionieren

JavaScripts Art der Objektorientierung ist anders als die anderer Sprachen. Daran hat auch die Einführung von Klassen nichts geändert, obwohl es auf den ersten Blick so aussehen mag. So funktioniert das ungewöhnliche System.

Von Lars Hupel

JavaScript geht bei Klassen und Objekten einen Sonderweg. Das dürfte viele Programmierer nicht überraschen, die Sprache ist bekannt für ihre zahlreichen exzentrischen Eskapaden, die – je nach Kontext – erheitern oder frustrieren. Ein gutes Beispiel ist das kuriose Verhalten des Gleichheitsoperators `==`:

```
 [] ==  0;   // -> true
 0  == "0";  // -> true
"0" == [];   // -> false
```

Ein leeres Array ist gleich 0 (aha?!), 0 ist gleich der Ziffer Null (okay …), aber die Ziffer Null ist nicht gleich dem leeren Array (soviel zur „Gleichheit“ des Operators). Man kann derart unintuitive Sperenzchen als Designfehler ansehen und im konkreten Fall den neueren, typprüfenden Vergleichsoperator `===` nutzen. Das hilft aber nur bedingt aus dem Schneider, denn JavaScript hat solche alten Zöpfe wie `==` nie abgeschnitten. Diese Rückwärtskompatibilität trug durchaus zur weiten Verbreitung der Sprache bei: Code, der vor Urzeiten im Netscape Navigator lief, läuft auch heute noch in Firefox, Chrome und diversen anderen JavaScript-Engines in Front- und Backend. Aber diese Kompatibilität bedeutet auch, dass man die alten Tricks und Tücken kennen muss, weil sich auch moderner Code nicht von ihnen freimachen kann.

Klasse oder Funktion?

Das gilt besonders für JavaScripts Objektsystem: Zahlreiche Programmierer frohlockten, als die Sprache im Jahr 2015 mit ECMAScript 6 das Schlüsselwort `class` einführte. Damit konnte man endlich komfortabel Klassen definieren und JavaScript fand Anschluss an alte Hasen der Objektorientierung wie Java und C++:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Nie wieder Objekte aus Funktionen erzeugen und Vererbung mit „Prototypen“ umsetzen, seltsamen Konstruktionen, die außer JavaScript kaum eine Sprache kannte? Nicht ganz, denn auch JavaScripts Klassen sind nur ein Zusatz, der die alten Wege weder abschafft noch überflüssig oder unsichtbar macht:

```
typeof Rectangle  // -> 'function'
```

Die vermeintliche Klasse `Rectangle` entpuppt sich als Funktion. Was ist hier los?

JavaScript hat – im Gegensatz zu beispielsweise TypeScript – keinen Compiler, der statisch Typen von Funktionen oder Variablen prüft. Stattdessen gibt es zwei Kategorien von Werten, die zur Laufzeit vorliegen können. Zum einen sind das skalare Werte (auch primitive Werte genannt): String, Number, BigInt, Boolean, Undefined, Symbol und Null. Solche Werte sind unveränderlich, es ist also beispielsweise nicht möglich, ein Zeichen an einer bestimmten Stelle in einem String zu modifizieren, ohne einen komplett neuen zu erstellen.

Die zweite Kategorie – also alles andere – sind Objekte. Der `typeof`-Operator bestimmt, welchen Typ ein Wert aktuell hat:

```
typeof "hi";  // -> 'string'
typeof 3;     // -> 'number'
typeof {};    // -> 'object'
```

Objekte sind also in JavaScript bloß Ansammlungen von Eigenschaften mit Werten, die sich ändern dürfen. Als Wert eignet sich jeder beliebige Typ, wodurch man Objekte auch verschachteln kann. Funktionen sind auch Objekte – eben Objekte, die man aufrufen kann.

Literale

Im Unterschied zu vielen anderen Programmiersprachen kann man in JavaScript konkrete Objekte erzeugen, indem man einfach ihre Eigenschaften in den Code schreibt. Um ein Objekt zu definieren, das die Maße eines Rechtecks repräsentiert, muss man also keine Klassen oder Konstruktoren programmieren, sondern notiert einfach ein Objektliteral in geschweiften Klammern:

```
let myRectangle = {
  width: 2,
  height: 4
};
```

Diese Syntax erlaubt es auch, einem Objekt Methoden zu verpassen. Die definiert man genauso wie jede andere Eigenschaft:

```
let myRectangle = {
  width: 2,
  height: 4,
  area: function() {
    return this.height * this.width;
  }
};
```

Der Aufruf funktioniert wie von anderen Sprachen gewohnt (`myRectangle.width` oder `myRectangle.area()`), lediglich die genaue Bedeutung von `this` ist trickreich (siehe Kasten „What's this?“ auf Seite 146).

Schablonen

Objektliterale sind praktisch, wenn man ein bestimmtes individuelles Objekt definieren will. Aber was, wenn man eine Art Schablone braucht, um viele ähnliche Objekte zu generieren? Genau dafür haben die meisten Programmiersprachen Klassen, aber JavaScript zeigt, dass es auch ohne geht:

```
function createRect(height, width) {
  return {
    width: width,
    height: height
  };
}
```

Um Objekte aus einer Schablone zu erzeugen, definiert man schlicht eine Funktion, die Parameter nimmt und ein per Literal definiertes Objekt zurückgibt.

Allerdings sieht man dem Resultat nicht an, aus welcher Funktion es stammt. Code, der so ein Objekt übergeben bekommt, kann ihm also nicht ansehen, dass es von `createRect()` kommt, ein Rechteck repräsentiert und folglich die Eigenschaften `width` und `height` hat.

In traditionellem JavaScript löst man diese Probleme, indem man Funktionen auf eine besondere Art schreibt und aufruft:

```
function Rect(height, width) {
  this.height = height;
  this.width = width;
}

let r = new Rect(20, 30);
r.width;  // -> 30
```

Das sieht auf den ersten Blick kaum anders aus, aber im Detail hat sich einiges getan: Statt explizit ein Objekt zu erzeugen, weist man die gewünschten Eigenschaften einfach `this` zu. Einen expliziten Rückgabewert gibt es nicht mehr. Außerdem ruft man die Funktion über den Operator `new` auf. Insgesamt erinnert die Syntax stark an Klassendefinitionen und Objektkonstruktionen in anderen Programmiersprachen; auch, weil solche Funktionen per Konvention Namen bekommen, die mit einem Großbuchstaben beginnen.

Aber wie ermöglicht diese Art der Objekterzeugung Vererbung, wie identifiziert sie, woher ein Objekt stammt und wie funktioniert so eine klassenartige Funktion überhaupt?

Prototypen

Zum einen enthält jedes Objekt eine interne Referenz auf ein sogenanntes Prototyp-Objekt. Auch der Prototyp ist ein Objekt, verweist also seinerseits wiederum auf ein Prototyp-Objekt und so weiter. Die Prototypenkette endet mit einem Verweis auf `null`. Über die eingebaute Methode `Object.getPrototypeOf(obj)` kann man die Kette nachvollziehen.

Zum anderen sind Funktionen in JavaScript eine spezielle Art von Objekt und haben daher ebenfalls ein Prototyp-Objekt. Zusätzlich stattet JavaScript aber praktisch jede Funktion mit noch einem weiteren Objekt aus, das in der Eigenschaft `prototype` steckt. Es gibt zwar ein paar Ausnahmen, aber die würden den Rahmen des Artikels sprengen.

Im Falle von `Rect` sind zwar beide Objekte leer, aber nicht identisch:

```
Object.getPrototypeOf(Rect);  // -> {}
Rect.prototype;  // -> {}
Object.getPrototypeOf(Rect) === ↵
        ↳ Rect.prototype;  // -> false
```

Das liegt daran, dass eine in `Rect.prototype` deklarierte Eigenschaft nicht in `Rect` selbst zur Verfügung steht, sondern in Objekten, die über `new Rect` entstehen. `Rect.prototype` ist in dem Sinne gewissermaßen die „Schablone“ für Objekte, die

c't kompakt

- JavaScript benutzt sogenannte Protoypen, um die Struktur von Objekten festzulegen, sie voneinander erben zu lassen und ihnen Funktionen zuzuordnen.
- Auch die mittlerweile breit unterstützte `class`-Syntax beruht auf diesem System.
- Um alten oder vertrackten Code zu verstehen, muss man die Eigenheiten von Prototypen kennen.

aus `new Rect` herausfallen. Es handelt sich also um zwei verschiedene Konzepte, die sich sehr ähneln.

Intern passieren außerdem zwei weitere Dinge, wenn man das `new`-Schlüsselwort vor dem Aufruf von `Rect` nutzt. Zuerst wird die in JavaScript eingebaute Funktion `Object.create()` aufgerufen, mit dem Objekt in `Rect.prototype` als Parameter. Der Aufruf erzeugt ein neues Objekt und dieses Objekt bekommt als Prototyp das Objekt in `Rect.prototype`. Zum anderen erhält das neue Objekt die Eigenschaft `constructor` mit einer Referenz zurück auf `Rect`. So merkt sich die Engine, welche Funktion das Objekt konstruiert hat.

Im zweiten Schritt wird die erstellende Funktion (im Beispiel `Rect()`) aufgerufen, wobei `this` auf das eben erzeugte neue Objekt verweist, das man dadurch mit Eigenschaften befüllen kann. Nun ist das neue Objekt fertig. Diese internen Schritte von `new` kann man auch manuell nachbauen:

```
let s = Object.create(Rect.prototype);
Rect.call(s, 20, 40);
s instanceof Rect;  // -> true
s.width;  // -> 40
s.constructor;  // -> [Function: Rect]
Object.getPrototypeOf(s) === ↵
        ↳ Rect.prototype;  // -> true
```

`new` nachzubauen ist in der Praxis zwar sinnlos, hilft aber zu verstehen, wie JavaScript funktioniert. Was der Aufruf `call()` tut, ist im Kasten „What's this?“ auf dieser Seite unten erklärt. Der `instanceof`-Operator prüft im Prinzip dasselbe wie die letzte Zeile. Er belässt es aber nicht beim ersten Prototyp-Objekt, sondern wandert die gesamte Prototypenkette entlang, um eine Übereinstimmung zu finden. Man könnte theoretisch die Prüfung umgehen, aber das ist ein Thema für einen anderen Artikel.

Methoden und Vererbung

Die Prototypenkette ersetzt in JavaScript die Klassenhierarchie anderer Sprachen: Versucht man auf eine Eigenschaft zuzugreifen, die ein Objekt nicht hat, dann schlägt die Engine sie automatisch im Prototyp-Objekt nach. Falls auch das scheitert, konsultiert die Engine den Prototyp des Prototyps und so weiter. Nur falls sie das Ende der Kette erreicht, ohne die Eigenschaft zu finden, ist das Objekt `undefined`.

Auf diese Weise kann man allen Rechtecken gemeinsame Eigenschaften bereitstellen, zum Beispiel eine Methode zur Flächenberechnung:

```
Rect.prototype.area = function() {
  return this.height * this.width;
}
```

Auf sämtlichen Objekten, die per `new Rect(…)` erzeugt worden sind (oder über den manuellen Nachbau), kann man nun `area()` aufrufen:

```
r.area();  // -> 600
```

What's this?

Kaum ein Schlüsselwort in JavaScript wird so oft missverstanden wie `this`. Wie in anderen Sprachen dient es dazu, in einer Methode das eigene Objekt zu referenzieren. Allerdings hat JavaScript eigentlich keine Methoden, sondern nur Funktionen.

Manche Funktionen sind als Eigenschaft an ein Objekt gebunden, sodass man sie per Punkt- oder Klammer-Schreibweise (`obj.f()` beziehungsweise `obj["f"]()`) aufrufen kann. Auch dieser Text spricht dann von „Methoden“, obwohl es sich streng genommen um normale Funktionen und nicht um Objektmethoden handelt: Die Bindung ans Objekt ist dynamisch.

Das Schlüsselwort `this` funktioniert in JavaScript daher anders: Beim Aufruf einer Funktion `f` bindet die JavaScript-Engine `this` dynamisch an dasjenige Objekt, auf dem `f` aufgerufen wird. Wenn im Code `obj.f()` steht, definiert also diese Zeile – und nicht der Ort, an dem `f` steht –, dass sich `this` auf `obj` bezieht.

Aus dieser dynamischen Zuordnung folgt auch, dass `this` in JavaScript nicht immer definiert ist:

```
let obj = {
  f: function() { return this; }
};
let g = obj.f;
g();  // -> undefined
```

`g = obj.f` gibt der Funktion einen neuen Namen, unabhängig von `obj`. Wenn man dann bloß `g()` aufruft, gibt es kein Objekt, dem `this` zugeordnet werden kann und es bleibt daher undefiniert. (Zumindest in JavaScripts „strict mode“. Im normalen, auch „sloppy“ genannten Modus liefert `this` in solchen Fällen das globale Objekt.)

Man kann die Referenz von `this` nicht nur kaputt machen, sondern auch umdefinieren:

```
g.call(obj)  // -> { f: [Function …] }
```

Statt `call()` kann man auch `apply()` aufrufen, der Unterschied besteht lediglich darin, ob man die eigentlichen Parameter für die Funktion – wenn es denn welche gibt – separat oder als Array übergibt. Wohlgemerkt erlauben die beiden Methoden es, eine Funktion auf einem beliebigen Objekt aufzurufen:

```
g.call(Math)  // -> Object [Math] {}
```

Damit so etwas nicht aus Versehen passiert, sieht man in JavaScript-Code manchmal folgendes Muster:

```
let h = obj.f.bind(obj)
h();  // -> { f: [Function …] }
```

Die Methode `bind()` erstellt eine Kopie ihrer Funktion (hier `f`), deren `this` fest an das übergebene Objekt (`obj`) gebunden ist. Dadurch kann man `h` auch ohne Objekt aufrufen und bekommt dennoch `obj` zurückgegeben.

Es gibt noch einige Randaspekte von `this` in JavaScript. Zum Beispiel unterstützt JavaScript seit Sprachversion 6 eine weitere Art, Funktionen zu definieren:

```
let obj = { f: () => { return this; }}
```

Mit dieser „fat arrow“-Syntax wird `this` nicht an die Funktion gebunden, sondern kommt aus dem umgebenden Kontext. Die JavaScript-Dokumentation von Mozilla erklärt diese und weitere Details (siehe ct.de/yaqr).

Das gilt auch für Objekte, die erzeugt wurden, noch bevor `area()` überhaupt definiert war, denn die Prototypenkette wird bei jedem einzelnen Zugriff durchlaufen. Prototypen sind sozusagen die – veränderlichen – Blaupausen für ganze Objektgruppen, denen man so gemeinsames Verhalten beibringt.

Über die Kette aus Prototypen funktioniert auch die Vererbung von Eigenschaften. Als Beispiel dient eine „Klasse" für allgemeine Formen, nicht nur Rechtecke:

```
function Shape(type) {
  this.type = type;
}
```

Nun kann man die Rechteck-Funktion um einen manuellen Aufruf der Eltern-Funktion erweitern:

```
function Rect(height, width) {
  Shape.call(this, "rectangle");
  this.height = height;
  this.width = width;
}
```

Anschließend muss man die Prototypenkette passend knüpfen. Das geschieht mit der eingebauten Funktion `Object.setPrototypeOf()`:

```
Object.setPrototypeOf(Rect.prototype,
                      Shape.prototype)
```

Der Prototyp für Rechtecke (gespeichert in der `prototype`-Eigenschaft von `Rect`) erhält also selbst als Prototyp das Prototyp-Objekt für allgemeine Formen (gespeichert in `Shape.prototype`).

Ein mittels `new Rect(…)` erzeugtes Rechteck `r` erhält `Rect.prototype` als Prototyp-Objekt. Es wird sozusagen vorn an die Kette angehängt, die der Interpreter bei jedem Aufruf durchläuft. Dadurch kann man auf `r` Funktionen aufrufen, die direkt als Eigenschaften von `r` definiert sind; Funktionen, die in `Rect.prototype` definiert sind und auch Funktionen, die `Shape.prototype` beisteuert.

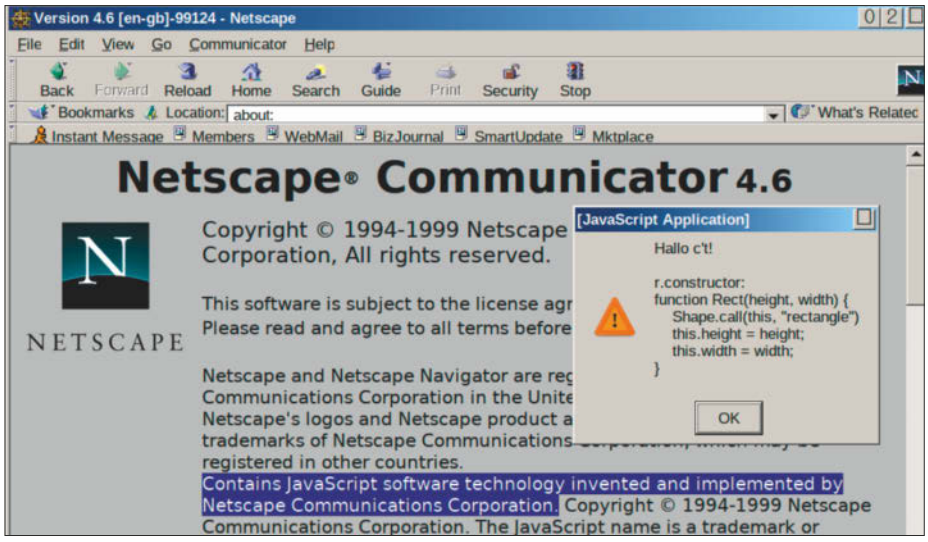
Die Infografik auf Seite 149 illustriert die verschiedenen Objekte und ihre Beziehungen zueinander. Schon dieses einfache Beispiel zeigt, warum Klassen in JavaScript sehnlich erwartet wurden: prototypenbasierte Vererbung ist ein sehr mächtiges Werkzeug, aber auch kompliziert und fehleranfällig.

Sogenannte Klassen

Mittlerweile unterstützt JavaScript auch Klassen und klassenbasierte Vererbung. Die Syntax dafür ist deutlich einfacher, übersichtlicher und vor allem ähnlich der Syntax anderer Programmiersprachen:

```
class Rectangle extends Shape {
  constructor(height, width) {
    super("rectangle");
    this.height = height;
    this.width = width;
  }

  area() {
    return this.height * this.width;
  }
}
```



JavaScript hat sich seit Netscapes Zeiten weit verbreitet. Das Objektsystem hat auch damals schon so wie heute funktioniert.

Lassen Sie sich aber nicht davon täuschen, dass so ein Codeschnipsel stark an andere verbreitete Sprachen erinnert: Die Klassen stellen lediglich eine eingängigere Syntax dar, darunter liegen nach wie vor Funktionen und verkettete Prototyp-Objekte:

- Die Klasse `Rectangle` ist in Wahrheit eine (spezielle) Funktion, deren Code man im `constructor` angibt. Allerdings erlaubt JavaScript nicht, diese Funktion ohne `new` aufzurufen – im Gegensatz zur klassischen Syntaxvariante.
- `extends` verdrahtet die Prototyp-Objekte in `Rectangle.prototype` und `Shape.prototype`, so wie ein Aufruf von `Object.setPrototypeOf()` es täte. (Zusätzlich verdrahtet das Schlüsselwort auch die Prototypen der Klassen selbst, sodass statische Eigenschaften ebenfalls vererbt werden: `Object.setPrototypeOf(Rectangle, Shape)`.)
- `super(…)` fasst den Aufruf des Konstruktors der Elternklasse in eine herkömmlichere Form als `Shape.call(this, …)`.
- `area()` und andere so spezifizierte Funktionen werden automatisch auf `Rectangle.prototype` definiert, ohne dass man dieses Objekt direkt modifizieren muss.

Es spricht nichts dagegen (und viel dafür), die Klassensyntax bei der Programmierung von JavaScript zu benutzen. Weil sie deutlich eingängiger ist, aber auch, weil sie Zugriff auf weiterführende Features bietet, etwa private Eigenschaften. Man darf dabei aber weder vergessen noch ignorieren, auf was für ein System JavaScript die Klassensyntax abbildet. Denn große Mengen existierender JavaScript-Code wurden noch nicht auf Klassen portiert und so mancher Code nutzt JavaScripts System in Konstruktionen, die man mit Klassen gar nicht umsetzen kann. Zum Beispiel funktioniert so manche eingebaute Funktion auch ohne `new` – tut dann aber nicht unbedingt dasselbe wie mit dem Schlüsselwort:

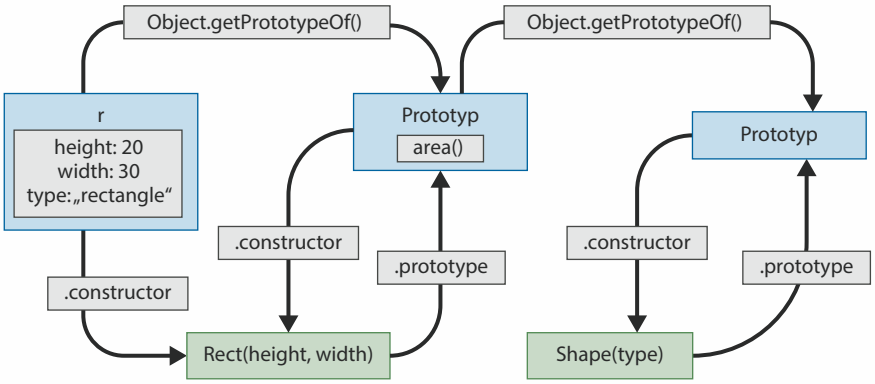
```
// Aufrufe als Konstruktor
new Array(1, 2, 3);  // -> [ 1, 2, 3 ]
new Date('1983')  // -> 1983-01-01…

// Aufrufe als Funktionen
Array(1, 2, 3); // -> [ 1, 2, 3 ]
Date('1983')  // -> 'Tue Sep 27 2022 …
```

`Array` liefert auch dann Arrays, wenn man es als Funktion aufruft. Auch `Date` funktioniert als Funktion, ignoriert in diesem Fall aber alle Parameter und gibt statt einem Datumsobjekt einen String mit dem aktuellen Datum zurück.

Prototypen-Vererbung

Das Objekt r wurde über die Funktion Rect() erzeugt, die wiederum Shape() aufruft, wodurch r alle Eigenschaften erhält, die Rect() und Shape() definieren. Durch eine passende Verdrahtung nach der Objekterzeugung entsteht eine Kette aus Prototyp-Objekten, über die sich Eigenschaften wie die Methode area() vererben lassen. Im Gegensatz zu klassenbasierter Vererbung, können die Prototypen und ihre Verkettung jederzeit verändert werden.



Fazit

Objektorientierung in JavaScript wird oft missverstanden. Sie funktioniert anders als in anderen Sprachen, woran die mittlerweile verfügbaren „Klassen“ nichts geändert haben. Die sind zwar ein sehr komfortabler Weg, um Objekte zu erzeugen, aber auch ihnen liegen Prototypen zugrunde.

Neben den im Artikel gezeigten Funktionen gibt es noch so manche technische Finesse für Nischenprobleme. Beispielsweise kann man die Prototypenketten jederzeit modifizieren. Eine gute Anlaufstelle für weiterführende Informationen sind die MDN Web Docs (siehe ct.de/yaqr). Gängige Konstruktionen und Probleme können Sie aber mit der in diesem Artikel gegebenen Übersicht verstehen und lösen. *(syt@ct.de)* **ct**

***MDN Web Docs:* ct.de/yaqr**