Objektorientierung in JavaScript

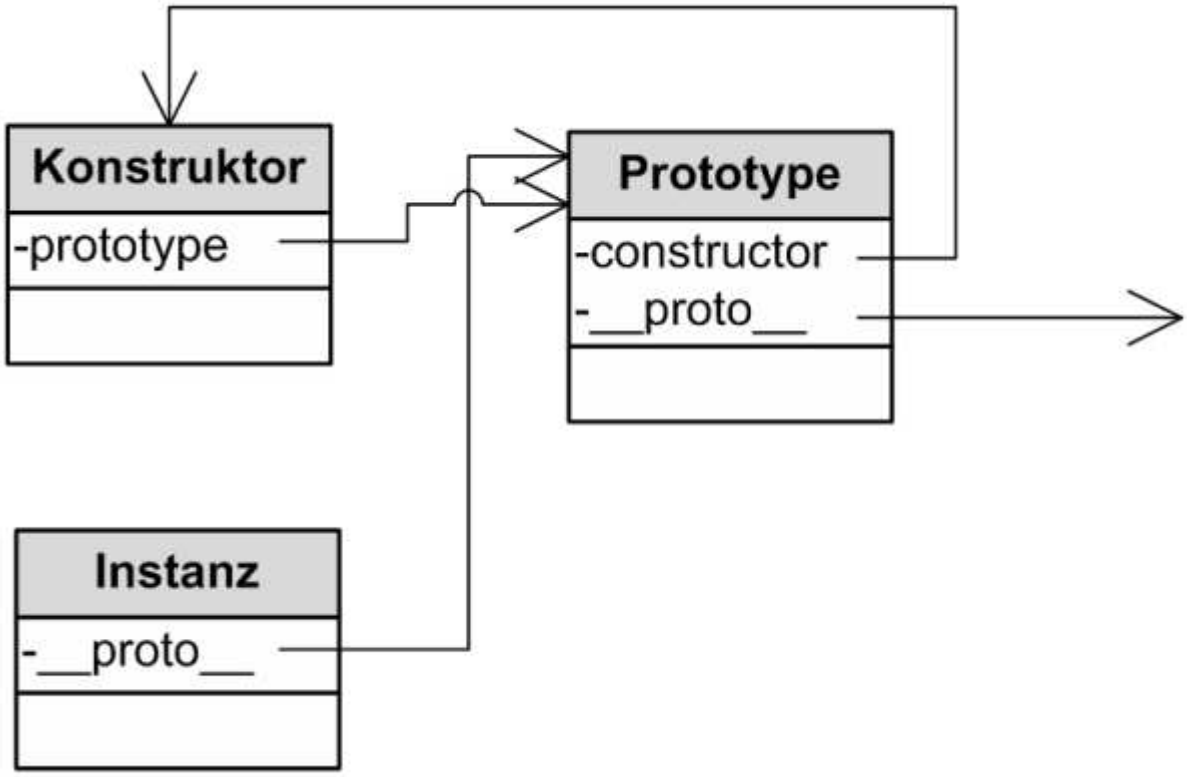
[Zurück] [Inhalt] [Themenübersicht] [Weiter]

Prototyping

Prototypen und die prototype-Eigenschaft

Alle Konstruktoren in JavaScript haben eine *prototype*-Eigenschaft, die automatisch erzeugt und initialisiert wird, wenn die Funktion definiert wird. Initial erhält diese Eigenschaft als Wert eine Referenz auf ein Objekt mit einer Eigenschaft, dem *constructor*-Attribut, welches auf den entsprechenden Konstruktor verweist. Dieses Objekt ist der Prototyp.

Jedes Objekt, das mit diesem Konstruktor erstellt wird, enthält unter der impliziten Eigenschaft *__proto__* eine interne Referenz auf diesen Prototypen.



Grafik 03: Zusammenspiel zwischen Konstruktor, Instanz und Prototyp

Alle Eigenschaften und Methoden dieses Prototyps sind auch Eigenschaften der Objekte für die es prototypisch ist, also für diejenigen Objekte, die mit dem in der *constructor*-Eigenschaft referenzierten Konstruktor erstellt wurden. Man kann sagen, dass ein JavaScript-Objekt alle Eigenschaften von seinem Prototyp über die *__proto__*-Eigenschaft erbt. Das Kapitel "Zugriff auf Objekt-Eigenschaften" zeigt warum das so ist.

Auch Prototypen enthalten die Eigenschaft *__proto__*. Worauf diese verweist und wie hierüber Eigenschaften vererbt werden, wird im Kapitel "Vererbung realisieren" gezeigt.

Das Prototyp-Objekt ist demnach für gemeinsam genutzte Eigenschaften und Methoden geeignet, da Objekte den gleichen Satz an Eigenschaften und Methoden von ihrem Prototyp erben.

Den Prototypen erweitern

In JavaScript ist es möglich dem Prototyp-Objekt weitere Attribute und Methoden hinzuzufügen. Diese Eigenschaften erscheinen, wie bereits erwähnt, als Eigenschaften aller Instanzen, die mit dem entsprechenden Konstruktor erstellt wurden. Dies gilt auch für bereits erstellte Objekte.

Attribute und Methoden werden dem Prototyp mit der bekannten DOT-Notation und einer Referenz auf die *prototype*-Eigenschaft gefolgt von einem Eigenschaftsnamen, sowie der Deklaration hinzugefügt.

Im Folgenden wird dem *Rechteck*-Prototyp (des Konstruktors aus vorangehenden Beispielen) eine Methode zur Flächenberechnung hinzugefügt:

```
//Methode flaeche zum Prototyp-Objekt hinzufügen
Rechteck.prototype.flaeche = function() { return this.breite * this.hoehe;}
```

Diese Methode kann nun von allen Instanzen von *Rechteck* wie eine objekteigene Methode genutzt werden.

```
r1 = new Rechteck(2,4);
alert(r1.flaeche());
```

Die dem Prototyp hinzugefügte Methode besitzt die Eigenschaften einer *static*-Methode in Java, also einer Methode, die von allen Objekten gemeinsam genutzt wird.

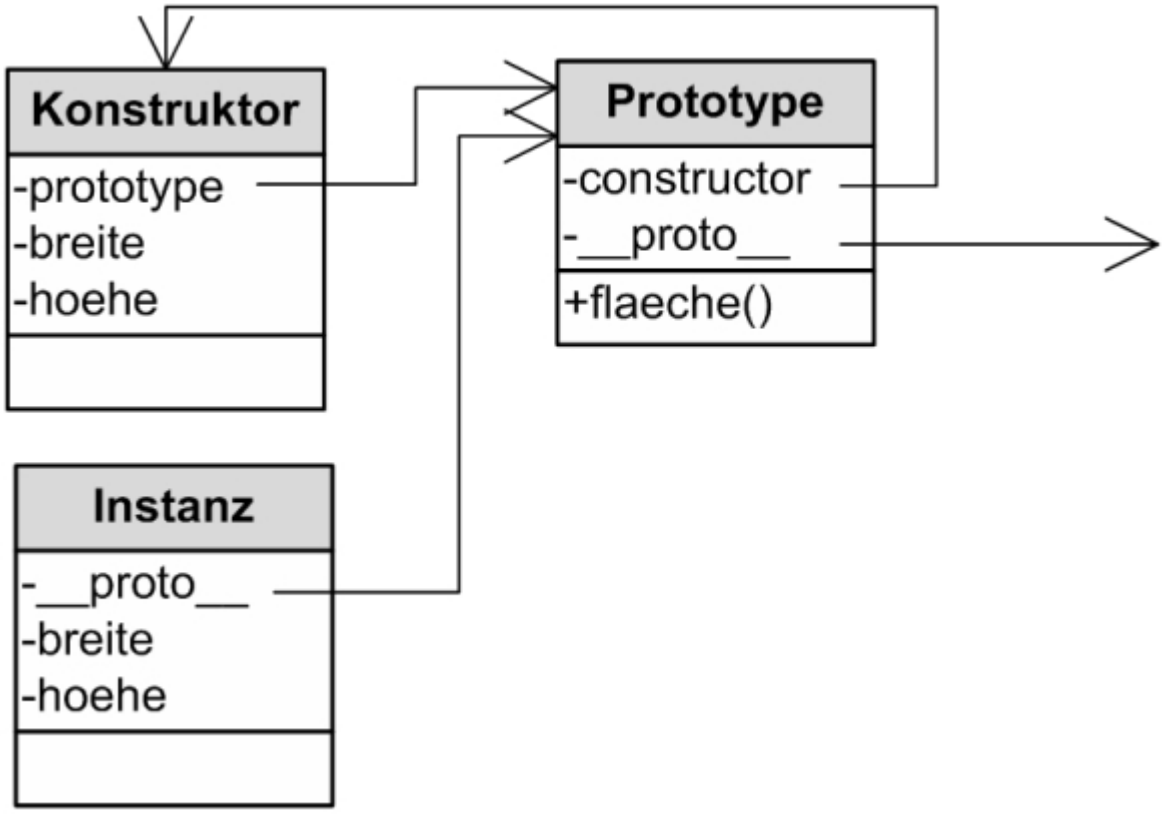
Hätte man diese Methode im Konstruktor deklariert, würde jedes Objekt diese Methode einzeln implementieren.

```
// Konstruktor
function Rechteck(w, h) {
  this.breite = w;
  this.hoehe = h;
  this.flaeche = function() { return this.breite * this.hoehe; }
}
```

Dies würde unnötig Speicherkapazität belegen und ist softwareergonomisch nicht sinnvoll.

Lokale und geerbte Eigenschaften

Die Eigenschaften, die dem Prototyp hinzugefügt werden, werden nicht etwa in die Instanzen kopiert, sondern bleiben Eigenschaften des Prototyps:



Grafik 04: Unterschied zwischen lokalen und geerbten eigenschaften

breite und *hoehe* sind direkte Eigenschaften der Instanz, da sie durch den Konstruktor in der Instanz angelegt wurden. Man bezeichnet sie daher als lokale Eigenschaften.

Die Methode *flaeche()* wurde dem Prototyp hinzugefügt und wird vom Prototyp an jede Instanz vererbt. Daher bezeichnet man sie als geerbte Eigenschaften.

Mit der *hasOwnProperty*-Methode lässt sich feststellen, ob geerbte oder lokale Attribute oder Methoden vorliegen:

```
//Button 1: Testet ob Methode flaeche in Prototyp vorhanden ist
alert(r1.__proto__.hasOwnProperty("flaeche"));
//Button 2: Testet ob Methode flaeche in Objekt vorhanden ist
alert(r1.hasOwnProperty("flaeche"));
//Button 3: Testet ob Attribut breite in Prototyp vorhanden ist
alert(r1.__proto__.hasOwnProperty("breite"));
//Button 4: Testet ob Attribut breite in Objekt vorhanden ist
alert(r1.hasOwnProperty("breite"));
//Button 5: Testet ob Attribut hoehe in Prototyp vorhanden ist
alert(r1.__proto__.hasOwnProperty ("hoehe"));
//Button 6: Testet ob Attribut hoehe in Objekt vorhanden ist
alert(r1.hasOwnProperty("hoehe"));
```

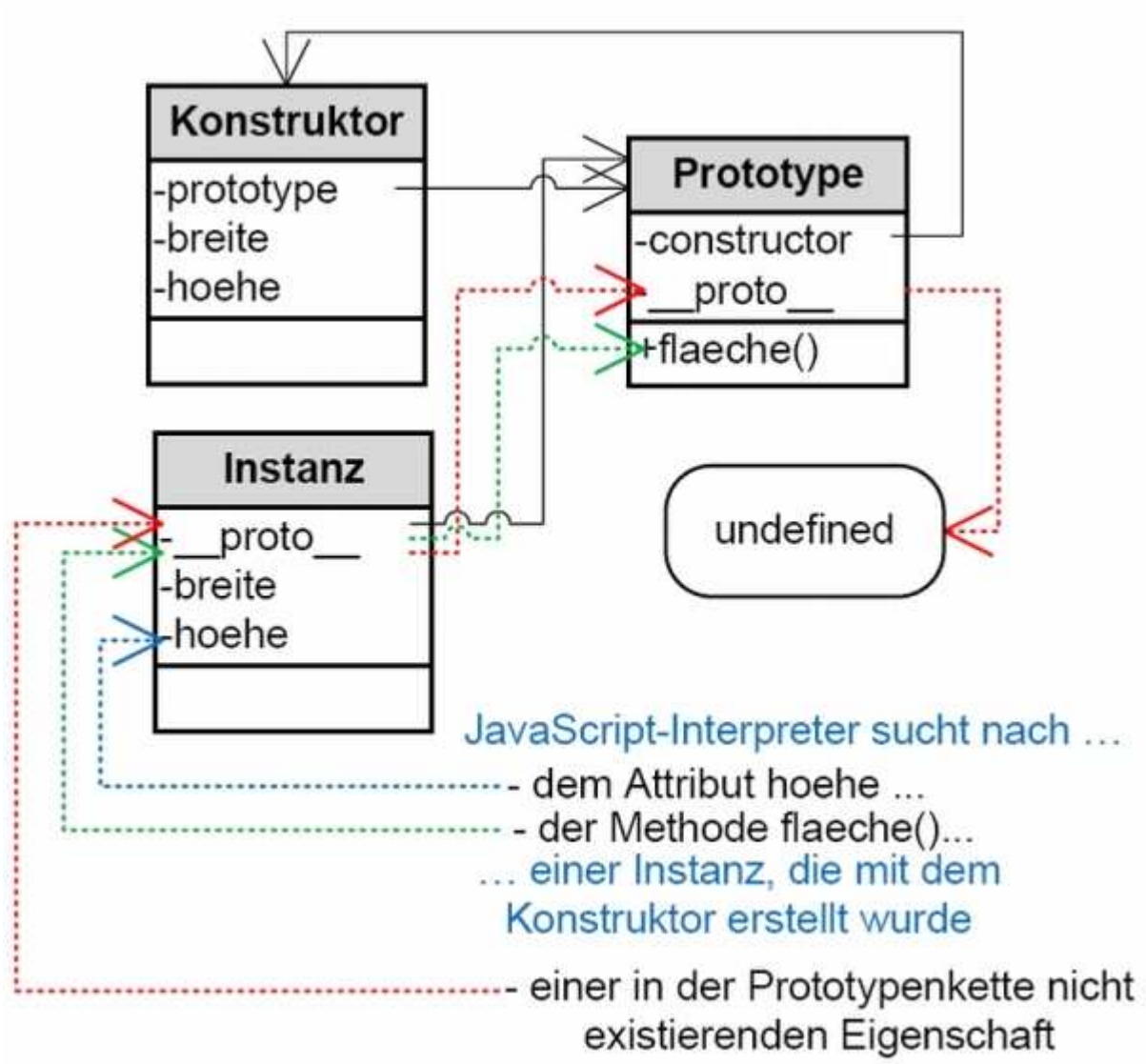
Button 1 Button 2 Button 3 Button 4 Button 5 Button 6

Zugriff auf Objekt-Eigenschaften

Zwischen dem Lesen und Schreiben von Eigenschaftswerten in JavaScript besteht eine Asymmetrie. Der Grund hierfür ist die Prototypenkette, die eine lineare Objekthierarchie darstellt. Wie man eine solche Prototypenkette erstellt, wird im nachfolgenden Kapitel gezeigt.

Wenn die Eigenschaft eines Objekts ausgelesen werden soll, prüft der Interpreter zunächst, ob diese Eigenschaft im Objekt vorhanden ist. Ist dies nicht der Fall, so wird weiterhin geprüft, ob diese Eigenschaft im (*__proto__*-Attribut referenzierten) Prototyp-Objekt enthalten ist oder dessen Prototyp-Objekt und so fort. Wird sie in dieser Prototypenkette gefunden, wird der entsprechende Wert zurückgeliefert, ansonsten *undefined*.

In Bezug auf das Rechteck-Beispiel verdeutlicht die folgende Grafik den Sachverhalt:



Grafik 05: Visualisierte Suche des Interpreters nach Objekteigenschaften

Die Eigenschaft *hoehe* wird als lokale Eigenschaft der Instanz direkt gefunden. Die Methode *flaeche()* hingegen wird im Prototyp, der über die *__proto__*-Eigenschaft der Instanz angesprochen wird, gefunden. Die *__proto__*-Eigenschaft dient also dem "lookup" von Eigenschaftswerten in der Prototypenkette beim lesenden Zugriff.

Bei dem Schreiben von Eigenschaftswerten verwendet JavaScript das Prototypen-Objekt nicht, denn hierdurch würden die Werte von allen durch diesen Prototyp abgeleiteten Objekten geändert werden. Stattdessen wird direkt im Objekt eine lokale Eigenschaft angelegt und mit dem zu schreibenden Wert belegt, sodass diese Eigenschaft nicht mehr vom Prototyp geerbt ist. Ist die Eigenschaft bereits lokal vorhanden, wird nur der Wert der Eigenschaft überschrieben. Rückwirkend wird dann beim Lesen dieser Eigenschaft auch nicht mehr auf das Prototyp-Objekt zurückgegriffen. Ebenso können auch Methoden dynamisch zu Objekten hinzugefügt werden.

Daraus ergibt sich, dass der Prototyp ideal für Methoden, die von allen Objekten gemeinsam genutzt werden (wie static-Methoden in Java) und konstante Eigenschaften, sowie für die häufige Nutzung von default-Werten ist.

[top]