Objektorientierte Konzepte – Prototyping

Lorenzo Sorressa (369509) und Karsten Bott (366619) Betreuer: David Korzeniewski

Einleitung

Diese Ausarbeitung zum Thema objektorientierte Konzepte soll dem Leser einen Einblick in ein Objektmodell verschiedener Programmiersprachen geben, welches sich vom klassenbasierten Konzept von Java unterscheidet. Hierbei werden die Grundlagen dieses Objektmodells beispielhaft an der Programmiersprache JavaScript erklärt, welche über ein prototypbasiertes Konzept verfügt.

Wir beginnen in Kapitel 1 mit der Objektdeklaration und allgemeinen Definition in JavaScript. Wir sprechen dabei über grundlegende Eigenschaften und Möglichkeiten eines Objekts in dieser Programmiersprache. Da JavaScript ein dynamisches Objektmodell besitzt, behandeln wir dieses in unserer Ausarbeitung in einem eigenen Kapitel. In Kapitel 2 geht es vor allem um die Flexibilität der Objekte, wobei wir Vor- und Nachteile dieser leichten Änderbarkeit abwägen. Nachdem wir über die Flexibilität der Objekte Bescheid wissen, grenzen wir diese mit Hilfe von sogenannten *Sealed and Frozen Objects* ein.

Sobald wir genug Allgemeinwissen über das Objektmodell von JavaScript gesammelt haben, können wir uns den Prototypen, quasi dem Pendant der Klassen in Java, widmen und sehen uns den größten Unterschied zu den bisher bekannten Objektorientierten Konzepten an. Wir lernen das Vererbungsmodell, die Auswirkungen auf Änderungen des Prototyps und, unter anderem, die Prototypenkette kennen. Besonders wichtig ist auch die Frage, warum man Prototypen überhaupt verwendet.

Zum Schluss vergleichen wir das Objektmodel von JavaScript mit dem von Java und blicken dabei auch auf ein drittes Konzept, nämlich auf das von Python.

Unsere wichtigsten Quellen hierbei sind „JavaScript – Von Kopf bis Fuß“ (Eric Freeman und Elisabeth Robson), „The principles of object-oriented JavaScript“ (Nicholas C. Zakas) und „Object-oriented Programming with Prototypes“ (Günther Blaschek).

Das Objektmodell in JavaScript

JavaScript enthält ein Konzept für Objekte, welche sich sowie in anderen objektorientierten Sprachen, wie Java, verhalten. Sie besitzen als Eigenschaften Attribute und das Verhalten von Objekten wird häufig mit mehreren Methoden definiert. Daher können logisch zusammenhängende Eigenschaften und Methoden als Verbund aus Informationen des Objektes betrachtet werden.

Als Grundlage wird zuerst ein Objekt erstellt. Diesem werden dann Attribute und Methoden zugewiesen. Dabei ist zu erkennen, dass Methoden in JavaScript von der Struktur her nichts anderes sind als Eigenschaften. Der Unterschied besteht nur darin, dass der Datentyp einer Methode der Typ `function` ist. Zudem kann man beim Deklarieren auch schon Attribute festlegen, auch wenn diese noch keine Werte erhalten sollen. Dafür nutzt man den Wert `undefined`. Weitergehend kann man auch ein Attribut erzeugen, welches später mal auf ein anderes Objekt zeigen soll. Dafür nutzt man den Wert `null`, welcher auf das leere Objekt verweist.

Auch das Kopieren von Objekten funktioniert so wie in anderen objektorientierten Sprachen. Daher wird beim Kopieren von Objekten keine Kopie des Objektes und dessen Inhalte, sondern nur eine Referenz auf das zu kopierende Objekt erzeugt.

```
var meinAuto={
      farbe:"blau",
      kW:120
};
meinAuto.zeigeFarbe=function(){
      window.alert("Farbe " + this.farbe);
};
function lackiereAuto(einAuto,farbe){
      einAuto.farbe=farbe;
}
lackiereAuto(meinAuto,"rot");
meinAuto.zeigeFarbe();
```

Beispiel 1.1: *Auto*

An diesem Beispiel lässt sich das Objektmodell aus JavaScript gut erkennen. Zuerst wird ein Objekt `meinAuto` deklariert und erhält dabei gleich zwei Eigenschaften `farbe` und `kw`. Daraufhin wird eine Methode `zeigeFarbe` und anschließend die Methode `lackiereAuto` definiert. Anschließend wird für die Methode `lackiereAuto` das übergebene Objekt `meinAuto` für den Parameterwert `einAuto` und die neue Farbe übergeben. Dadurch kommt es für das Objekt `meinAuto` zur Änderung und Ausgabe des Attributes `farbe`.

Dynamisches Objektmodell

Das dynamische Objektmodell von JavaScript ist einer der größten Unterschiede im Vergleich zu Java. Gleichzeitig ist dieses Modell auch eine große Stärke und kann ein mächtiges Werkzeug für den Programmierer sein. Objekte können beliebig verändert, erweitert und angepasst werden, da sie als sogenannte *freie Objekte* keiner Klasse, bzw. Vorschrift untergeordnet sind.

```
var auto = {
            marke: "Volkswagen",
            baujahr: 2007,
            fahren: function(){
                  //Code zum Fahren
            }
};
```

Beispiel 2.1: *Auto*

In Beispiel 2.1 wird ein Objekt erstellt, welches ein Auto darstellen soll. Es besitzt zwei Attribute `marke` und `baujahr`, sowie eine Methode `fahren`. Das dynamische Objektmodell von JavaScript erlaubt es uns nun, im weiteren Verlauf des Codes, Attribute oder Methoden zum Objekt `auto` hinzuzufügen. Hierfür benutzen wir die Punktnotation (siehe Beispiel 2.2): Gibt es das Attribut, bzw. die Methode schon, wird der Wert geändert, ansonsten wird ein neues Attribut, bzw. eine neue Methode erzeugt.

```
//Ändern eines vorhandenen Attributs
auto.baujahr = 2008;

//Erweitern um ein neues Attribut
auto.motorAn = false;

//Erweitern um eine neue Methode
auto.motorStarten = function(){
      this.motorAn = true;
};
```

Beispiel 2.2: Auto

Nachdem das Objekt `auto`, wie in Beispiel 2.2 gezeigt, erweitert wurde, besitzt es nun drei Attribute und zwei Methoden. Ebenso kann man Attribute und Methoden mit dem Schlüsselwort `delete` wieder entfernen und so das Objekt weiter verändern.

Weitere Vorteile sind das einfache Hinzufügen und Entfernen von Attributen und Methoden. Falls bei der Initialisierung des Objekts nicht feststeht, welche Attribute es genau erhalten soll, können beliebig viele Attribute, auch zu einem späteren Zeitpunkt, hinzugefügt oder gelöscht werden.

Diese Anpassung von Objekten kann beliebig ausgedehnt werden. Es können jederzeit Attribute und Methoden hinzugefügt, geändert und wieder entfernt werden.

```
//Das Auto wird zum Flugzeug
delete auto.motorAn;
delete auto.motorStarten;
delete auto.fahren();
auto.marke = "Boeing";
auto.modell = "747";
auto.fliegen = function(){
      //Code zum fliegen
};
```

Beispiel 2.3: Auto

Die Flexibilität im dynamischen Objektmodell von JavaScript geht soweit, dass man durch Änderung der Eigenschaften wie hier im Beispiel 2.3 dargestellt, aus einem Auto ein Flugzeug werden kann. Es besitzt nun nur noch die Attribute `marke` und `modell`, sowie die Methode `fliegen`. Ein Vorteil dieser Implementierung kann sein, dass dasselbe Objekt im Verlauf des Codes verschiedene Aufgaben übernehmen kann. Im Beispiel könnte das Objekt `auto` zuerst ein Auto und später im Code ein Flugzeug darstellen.

Allerdings kann diese Flexibilität des Objektmodells auch Nachteile nach sich ziehen:

Wie im Beispiel 2.3 gesehen, kann die starke Änderung von Objekten auch zu Verwirrungen führen. Ein Objekt mit dem Namen `auto`, welches zum Zeitpunkt der Initialisierung auch ein Auto dargestellt haben kann, muss nicht zwangsläufig auch ein Auto bleiben. Dies führt unter Umständen zu Fehlern im Programm. Um diese Probleme zu vermeiden, kann mit Hilfe von `in` geprüft werden, ob ein Attribut oder eine Methode im Objekt enthalten ist. Beispielsweise evaluiert „`fahren in auto`“ im Beispiel 2.1 zu `true` und im Beispiel 2.3 zu `false`.

Im folgenden Kapitel befassen wir uns mit *Sealed and Frozen Objects*, die dazu verwendet werden können, die vorher eventuell nicht gewünschte Flexibilität des dynamischen Objektmodells zu unterbinden. [FR14]

Sealed and Frozen Objects

Um das dynamische Verhalten der Objekte einzuschränken, hält JavaScript die drei Möglichkeiten *preventExtensions, Sealed Objects* und *Frozen Objects* bereit. Jede dieser drei Möglichkeiten schränkt die Flexibilität der Objekte unterschiedlich ein, wobei die *Sealed Objects* auf den Einschränkungen von *preventExtensions* und die *Frozen Objects* auf den *Sealed Objects* aufbauen.

Die Methode `preventExtensions` (dt.: „Erweiterungen verhindern“), welche von *Object*[1] stammt, erhält als Parameter ein Objekt. Anschließend ändert sich dadurch das Attribut `isExtensible` des Objekts von `true` auf `false` und verhindert so, dass weitere Attribute oder Methoden hinzugefügt werden können. Es ist jedoch immer noch möglich Attribute und Methoden zu entfernen, sowie deren Werte zu ändern und zu lesen.

```
//Anwendung von preventExtensions
var auto = {
            marke: "Audi"
};
console.log(Object.isExtensible(auto)); //true
//Erweitern verhindern
Object.preventExtensible(auto);
console.log(Object.isExtensible(auto)); //false
//Versuch hund um eine neue Funktion zu erweitern
auto.fahren = function(){
      //fahren
};
//Versuch fehlgeschlagen
      console.log("fahren" in auto); //false
```

Beispiel 3.1: Auto

Darauf aufbauend operieren die sogenannten *Sealed Objects*. Bei einem *Sealed Object* (dt.: „versiegeltes Objekt“) können wie bei `preventExtensions`, keine neuen Attribute oder Methoden hinzugefügt werden und zusätzlich bereits vorhandene nicht gelöscht werden. Mit der Methode `seal`, welche als Parameter das gewünschte Objekt erhält, verwandelt man in JavaScript ein Objekt in ein *Sealed Object*. Hierbei wird der Wert des Attributs `isSealed` von `false` auf `true` geändert (siehe Beispiel 3.2).

Sealed Objects entsprechen den klassischen Objekten in objektorientierten Programmiersprachen, wie Java oder C++, da ihre Werte nur geändert und gelesen werden können.

[1] Object ist die oberste Hierarchie von Objekten, vergleichbar mit Object in Java

```
//Sealing Object
console.log(Object.isSealed(auto)); //false
//Objekt versiegeln
Object.seal(auto);
console.log(Object.isExtensible(auto)); //false
console.log(Object.isSealed(auto)); //true
//Versuch Attribut zu löschen
delete auto.marke;
//Versuch fehlgeschlagen
console.log("marke" in auto); //true
```

Beispiel 3.2: Auto

Des Weiteren existieren in JavaScript die *Frozen Objects* (dt.: „eingefrorene Objekte“). *Frozen Objects* sind *Sealed Objects* mit der zusätzlichen Eigenschaft, dass die Werte der Attribute und Methoden unveränderbar sind. Das entsprechende Objekt ist nun sinngemäß im aktuellen Zustand eingefroren, da keine Änderungen mehr möglich sind und die Eigenschaften nur gelesen werden können. Versucht das Programm den Wert eines Attributs zu ändern, passiert nichts (siehe Beispiel 3.3). Mit Hilfe der Methode `freeze`, die ähnlich wie `seal` funktioniert, wird das Attribut `isFrozen` von `false` auf `true` geändert und das jeweilige Objekt wird zu einem *Frozen Object*.

```
//Frozen Object
console.log(Object.isFrozen(auto)); //false
//Objekt einfrieren
Object.freeze(auto);
console.log(Object.isExtensible(auto)); //false
console.log(Object.isSealed(auto)); //true
console.log(Object.isFrozen(auto)); //true
//Versuch Attribut zu ändern
auto.marke = "Mercedes";
//Versuch fehlgeschlagen
console.log(auto.marke); //"Audi"
```

Beispiel 3.3: Auto

Auf diese Weise kann das dynamische Verhalten von Objekten der Aufgabe entsprechend angepasst und so mögliche Fehlerquellen verhindert werden. Nachteil dieser Maßnahmen ist, dass die vorgenommenen Einschränkungen nicht mehr rückgängig zu machen sind. Es können zwar zusätzliche Einschränkungen vorgenommen, aber keine vorhandenen wieder entfernt werden. [ZA14]

Das Vererbungsmodell – Prototyping

Bislang haben wir nur mit *freien Objekten* in JavaScript gearbeitet. Diese haben allerdings den Nachteil, dass keine einheitliche Modellierung von zusammenhängenden Objekten möglich ist. Beispielsweise können wir viele verschiedene Autos als *freie Objekte* erstellen, doch wir können nicht einheitlich erkennen, ob diese Objekte vom Typ `Auto` sind oder nicht, da die erstellten Objekte alle vom Typ `Object` sind. Außerdem können so die verschiedenen Autos unterschiedliche Attribute und Methoden zur Verfügung haben, wodurch die Kategorisierung noch schwieriger wird.

Hierfür stellt JavaScript die Objektkonstruktoren zur Verfügung, welche analog zu den Konstruktoren in Java funktionieren. Ein Konstruktor wird wie eine Methode deklariert und enthält die Eigenschaften, welche ein Objekt, das mit diesem Konstruktor erzeugt wird, erbt. Er kann so für die einheitliche Erstellung von Objekten gleichen Typs sorgen. Ein Konstruktor unterscheidet sich von anderen Methoden durch das Schlüsselwort `this`, welches auf das aktuelle Objekt verweist. Also das Objekt, welches vom Konstruktor erzeugt werden soll.

```
//Auto-Konstruktor
function Auto(marke, laenge, gewicht){
      //Parameterwerte initialisieren
      this.marke = marke;
      this.laenge = laenge;
      this.gewicht = gewicht;
      //Funktion deklarieren
      this.starten = function(){
            //starten
      };
};
```

Beispiel 4.1: Auto

Nach Erzeugung des Objekts mit dem Schlüsselwort `new` können zwar weitere Attribute und Methoden, wie bei den *freien Objekten*, hinzugefügt und entfernt werden, aber zum Zeitpunkt der Initialisierung sind alle Autos, bis auf den Inhalt ihrer Werte, gleich. Man kann nun sogar, mit Hilfe der Methode „`instanceOf(nameKonstruktor)`“, überprüfen, ob ein Objekt vom Typ `Auto` ist oder nicht. Ein Objekt ist nur vom Typ `Auto`, wenn es mit dem entsprechenden Konstruktor erstellt wurde, ansonsten nicht.

Bei der Erzeugung des Objekts mit einem Konstruktor werden alle Attribute und Methoden kopiert und im neuen Objekt abgespeichert. Dies führt dazu, dass die Methode mehrfach gespeichert wird. Allerdings soll die Methode `starten` für jedes `Auto` gleich funktionieren und wird daher nur einmal gebraucht. Dies kann einen vollen Speicher nach sich ziehen und zu Inkonsistenz im Code führen

Um dieses Problem zu lösen besitzt JavaScript die sogenannten *Prototypen*.

Prototypen können Attribute und Methoden beinhalten und gehören einem oder mehreren Konstruktoren an. Die Besonderheit bei Prototypen ist, dass Objekte, die von ihnen erben keine Kopie der Eigenschaften erhalten, sondern direkt auf die Implementierung im Prototyp zugreifen. Da Prototypen selbst wiederum Objekte sind, können ihre Attribute und Methoden direkt abgerufen werden. Bei der Modellierung unserer Autos sollten alle Eigenschaften, die alle Autos erhalten sollen, im Prototypen deklariert werden. In unserem Fall ist dies nur die Methode `starten`.

```
//Funktion aus Konstruktor entfernen
delete Auto.starten;
//Prototype-Objekt erstellen
Auto.prototype = {
      starten: function(){
            //starten
      }
};
```

Beispiel 4.2: Auto

In Beispiel 4.2 weisen wir dem Attribut `prototype` ein neues Objekt als Prototypen-Objekt zu und deklarieren in diesem die Methode `starten`. Die Methode kann nun von allen Objekten, welche mit Hilfe des Konstruktors `Auto` erzeugt wurden, aufgerufen werden. Sie wird dazu nicht mehr von den Objekten kopiert.

Allgemein sollte man jede Eigenschaft, welche in allen Objektinstanzen gleich implementiert sein muss in den Prototypen auslagern, um Mehrfachspeicherung zu vermeiden. Würde man in unserem Beispiel das Attribut `marke` auslagern und mit „`Audi`“ initialisieren, wären alle Autos von der Marke Audi. Wenn wir diesen Wert im Prototypen ändern, passen sich auch sofort alle Auto-Instanzen an, da sie nur auf das Attribut im Prototypen verweisen, es aber nicht selber implementieren. Möchte man aber, dass ein bestimmtes Objekt von der Marke Mercedes ist, so muss das entsprechende Objekt die Eigenschaft `marke` implementieren damit es so das gleichnamige Attribut im Prototyp überdeckt. Hierbei sucht das Programm erst im jeweiligen Objekt nach der Eigenschaft und erst danach im Prototyp. Dabei sind nur die innersten Eigenschaften, ähnlich wie in der Vererbungshierarchie von Java, für das Programm sichtbar. Überdeckte Attribute und Methoden werden nicht beachtet.

Prototypenkette

Im vorherigen Kapitel haben wir kennengelernt, wie man mit der Auslagerung von Eigenschaften in einen Prototyp Mehrfachspeicherung vermeidet. Allerdings haben wir immer noch Code, der mehrfach abgespeichert wird, da die Attribute `marke`, `laenge` und `gewicht` in jeder Instanz von `Auto` gespeichert werden. Mit Hilfe der sogenannten *Prototypenkette* können wir in JavaScript eine mehrteilige Vererbungshierarchie aufbauen und so Mehrfachspeicherung besser vermeiden. Da ein *Prototyp* ebenfalls ein Objekt ist, kann ein *Prototyp A* also problemlos von einem *Prototyp B* erben. Das bedeutet *Prototyp B* ist der *Prototyp* von *Prototyp A*.

Diese Art von Vererbung nennt man *Prototypenkette* und kann beliebig oft erweitert werden. Im folgenden Beispiel unterscheiden wir mit Hilfe von der Prototypenkette, ob es sich bei unseren Autos um einen Audi oder um einen Mercedes handelt.

```
//Prototypenkette
//Konstruktoren deklarieren
function Auto(){
      this.starten = function(){
            //starten
      };
};
function Audi(){
      this.marke = "Audi";
};
function Mercedes(){
      this.marke = "Mercedes";
};
function AudiAuto(laenge, gewicht){
      this.laenge = laenge;
      this.gewicht = gewicht;
};
function MercedesAuto(laenge, gewicht){
      this.laenge = laenge;
      this.gewicht = gewicht;
};
//Prototypen zuweisen
Audi.prototype = new Auto;
Mercedes.prototype = new Auto;
MercedesAuto.prototype = new Mercedes;
AudiAuto.prototype = new Audi;
```

Beispiel 5.1: Auto

Alle Tablets, die von `AudiAuto` erzeugt werden, erben von `Audi` und von `Auto`. Diese müssen nicht mehr das Attribut `marke` implementieren. [FR14]

Fazit

Vergleicht man die Objektorientierung der Sprachen Java und JavaScript stellt man fest, dass Objekte in JavaScript flexibler sind und vom Programmierer explizit eingeschränkt werden können. Dadurch wird dem Programmierer freigestellt, wie die Objekte im Code modelliert werden sollen, allerdings kann dies auch zu Fehlern führen. In Java stammen die Objekte direkt von einer Klasse ab und besitzen daher eine vorgegebene Struktur. In JavaScript kann eine solche Struktur, zumindest teilweise mit den Prototypen erreicht werden.

Dadurch kann in JavaScript eine ähnliche Konsistenz wie in Java erreicht werden. In Java können dagegen Objekte mit *Reflection* flexibler gemacht werden.

Beide Programmiersprachen unterscheiden sich zwar in den Details der Vererbung, aber mit beiden können ähnliche Programmierkonzepte erarbeitet werden.

In der Programmiersprache Python ist es möglich Attribute, wie in JavaScript auch, zu bereits initialisierten Objekten hinzuzufügen. Genau wie Java besitzt Python eine klassenbasierte Objektorientierung und hat daher eine konsistentere Grundmodellierung als JavaScript, da das Objektmodell sich ohne zusätzliche Änderungen auf ein Klassenmodell stützt. Dagegen muss bei JavaScript diese Konsistenz erst, mit Hilfe von Konstruktoren und Prototypen, nachträglich erreicht werden. Verschiedene objektorientierte Sprachen unterscheiden sich daher in der jeweiligen Implementierung und Vererbung von Objekten und können in der Umsetzung sehr unterschiedlich sein.

Quellen

[BW14] Bewersdorf, J.: Objektorientierte Programmierung mit JavaScript. Springer Verlag, Wiesbaden, 2014.

[BL94] Blaschek, G.: Object-Oriented Programming. Springer Verlag, Berlin, 1994.

[FR14] Freeman, E.; Robson, E.: JavaScript-Programmierung von Kopf bis Fuß. O'Reilly, Köln, 2014.

[KL14] Klein, B.: Python 3 Tutorial. Klassen. Stand: März 2014. URL: http://www.python-kurs.eu/python3_klassen.php (letzter Abruf: 17.05.2017)

[ZA14] Zakas, N. C.: The principles of object-oriented JavaScript. No Starch Press, San Francisco, 2014.