Foto: Shutterstock / kentoh

SCHNELLEINSTIEG IN JAVASCRIPT (TEIL 2)

Objektorientiert

In JavaScript kann man auch objektorientiert programmieren.

Im folgenden Artikel wird eine Einführung in die objektorientierte Programmierung mit JavaScript gegeben.

Die objektorientierte Programmierung galt bei ihrer Einführung als die Lösung für die immer komplexer werdenden Problemstellungen und deren Umsetzung in Programmen. Statt die reale Welt in Prozeduren und Funktionen zu transformieren, war es mit der objektorientierten Programmierung erstmals möglich, die Struktur der Wirklichkeit in Form eines Programms nachzubilden.

Anstatt also eine Person nur mit Funktionen und Prozeduren zu beschreiben beziehungsweise die einzelnen Daten wie zum Beispiel Nachname, Vorname etc. zu verarbeiten, ist es mit der objektorientierten Programmierung möglich, ein reales Objekt in Form eines Programms nachzubilden. Diese Art der Programmierung kommt dem menschlichen Denken näher als die prozedurale Programmierung.

So verfügt ein Objekt, zum Beispiel eine Person, über Eigenschaften (Name, Größe oder auch Personalnummer) und eben diese Eigenschaften lassen sich mit der objektorientierten Programmierung in Form von Programmcode nachbilden. Um es vorwegzunehmen: Auch die OOP (Abkürzung für objektorientierte Programmierung) war nicht die endgültige Lösung. Als Baustein trägt sie aber dazu bei, umfangreichen Anforderungen der realen Welt besser in Programmcode abzubilden.

Ein Objekt ist dabei der zentrale Dreh- und Angelpunkt während der Entwicklung. So kann ein Objekt nicht nur Daten enthalten, sondern es stehen auch Methoden/Funktionen innerhalb des Objektes bereit, um die enthaltenen Daten zu manipulieren. Ein Beispiel für ein Objekt ist ein Formular, das Sie zur Programmlaufzeit aufrufen. Dieses Formular besitzt viele unterschiedliche Eigenschaften (*Properties*), beispielsweise die Höhe (*Height)* und Breite (*Width)*. Natürlich können diese Eigenschaften beziehungsweise die darin enthaltenen Daten auch zur Laufzeit wieder geändert werden.

Auslöser der Änderungen ist in der Regel ein Ereignis (*Event*), das zu einem definierten Zeitpunkt eintritt. Betrachtet man es genauer, so verbirgt sich hinter einem Ereignis eine entsprechende Methode. Und nun ist die Definition beisammen. Ein Objekt ist, im Prinzip, die Variable eines ▶

Typs, der nicht nur Daten enthält, sondern auch Funktionen, um diese Daten zu bearbeiten. In der Begriffswelt der Objektorientierung spricht man allerdings nicht von Funktionen, sondern nur allgemein von den Methoden eines Objekts.

Klassen und Objekte

Da ja laut Definition im vorherigen Abschnitt Objekte nur die Variablen sind, sollte es natürlich auch (Daten-)Typen geben, von denen die Objekte abgeleitet werden. Hierfür gibt es in vielen OOP-Sprachen die Klasse, welche in der Regel mit dem Schlüsselwort *class* definiert wird. Eine Klasse ist eine Vorlage für ein Objekt. Die Sprache JavaScript allerdings kennt das Konstrukt der Klasse nicht.

Also ist eine Klasse so eine Art Schablone. Sie erinnern sich bestimmt noch an die Förmchen im Sandkasten. Die Form ist die Klasse und das Häufchen Sand ein Objekt. Im letzten Teil der Serie haben Sie Felder (Arrays) als ein Element zur Speicherung von Daten kennen gelernt. Ein Feld besteht aus einer bestimmten Anzahl von Variablen eines Datentyps.

Stellen Sie sich eine Klasse also (erst einmal) einfach wie ein Feld vor, mit dem Unterschied, dass dieses mehrere Variablen unterschiedlichen Datentyps beinhalten kann. Außer den Daten kann eine Klasse zusätzlich noch Programmcode enthalten. Dieser Programmcode wird in Bereichen abgelegt, die im Fachjargon als Methoden (in JavaScript nennt man diese Funktionen) bezeichnet werden.

JavaScript und Objekte

JavaScript kennt das Konstrukt Klasse nicht und es gibt auch kein Gegenstück. Allerdings kann JavaScript mit Variablen aller Art umgehen. Es hält Sie also nichts davon ab, in JavaScript objektorientiert zu programmieren. Die typischen Merkmale der Objektorientierung (wie zum Beispiel Klassen, Kapselung etc.) fehlen natürlich trotzdem. Da es das Schlüsselwort *class* in JavaScript nicht gibt, wird stattdessen *function* verwendet. In JavaScript muss sämtlicher Code, welcher geschrieben wird innerhalb einer Funktion eingeschlossen werden. Aus diesem Grund bietet sich dieses Konstrukt natürlich auch für das Anlegen einer Variablen mit den Eigenschaften einer Person an:

```
function Person() {
  this.Nachname = '-';
  this.Vorname = '-';
  this.Personalnummer = 0;
}
```

Die Variablen werden unter Angabe des Namens und des Schlüsselwortes *this* innerhalb der Funktion *Person* angelegt. Mit dem Schlüsselwort *this* können Sie einen Bezug zum aktuellen Objekt herstellen. Später folgen weitere Informationen zur Verwendung des Schlüsselworts *this*. In diesem speziellen Fall zur Anlage der drei Variablen. Initialisiert werden Sie in diesem Beispiel mit Standardwerten. Erst im folgenden Programm soll den einzelnen Variablen jeweils ein Wert zugewiesen werden. Dies geschieht in einer JavaScript-Funktion, welche einem HTML-Formular zugeordnet ist. Auch hier finden Sie einige Schlüsselworte, welche erst später erläutert werden. Wichtig ist der Teil des Programms, in welchem die neue Variable Person angelegt und initialisiert wird:

```
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
  //Quellcode entfernt
    var person = new Person();
    person.Nachname = 'Meier';
    person.Vorname = 'Hans';
    person.Personalnummer = 12345;
  };
})();
```

Hier werden die Variablen des Objekts über einen Punkt getrennt vom Objektnamen *person* aufgerufen. Auch hier ermöglicht der Punkt-Operator den Zugriff auf Bestandteile des Objekts.

Vielleicht haben Sie schon einmal gesehen, dass an der einen oder anderen Stelle in anderen Sprachen Variablen einer Klasse oder auch eine Klasse selbst mit einem zusätzlichen Schlüsselwort wie *public* oder *private* versehen sind. Mit diesen Schlüsselwörtern wird der Zugriff auf ein gekennzeichnetes Element gesteuert. Ist ein Element, beispielsweise eine Variable, die innerhalb einer Klasse definiert wurde, mit dem Schlüsselwort *public* gekennzeichnet, so ist ein Zugriff von außen möglich. Wird hingegen das Schlüsselwort *private* verwendet, so ist der Zugriff auf die Variable geschützt und von außen nicht möglich.

In JavaScript gibt es die Schlüsselworte *private* oder *public* nicht. Alle definierten Elemente einer Funktion sind somit erst einmal nach außen hin sichtbar. Trotzdem kann man durch Verwendung von Funktionen den Zugriff auf Variablen einschränken, wie das folgende Listing demonstriert:

```
function Konto() {
  this.Kontonummmer = "123456";
  function Geheimnummer() {
    var gNummer = 999999;
  }
}
```

Damit die Variable *gNummer*, welche die Geheimnummer enthält, nach außen hin nicht sichtbar ist, wird diese innerhalb der Funktion *Geheimnummer* mit Hilfe des *var* Schlüsselwortes angelegt. Dadurch, dass die Variable *gNummer* innerhalb der Funktion mit dem *var* Schlüsselwort angelegt wurde, ist sie direkt der Funktion *Geheimnummer* zugeordnet und so von außen nicht sichtbar. Mit folgendem Listing lässt sich das Ganze überprüfen:

```
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
  //Quellcode entfernt
  var mein_konto = new Konto();
```

```
  var test_kontonummer = mein_konto.Kontonummmer;
  var test_gNummer = mein_konto.gNummer;
  };
})();
```

Nach Anlage des neuen Objekts *mein_konto* wird in den beiden folgenden Zeilen versucht, die Variablen *Kontonummer* und *gNummer* auszulesen. Die Variable *Kontonummer* wurde mit dem Schlüsselwort *this* angelegt. Dadurch wird ein direkter Bezug zur Funktion *Konto* hergestellt und die Variable ist auch nach außen hin sichtbar. Die Zuweisung zur lokalen Variable funktioniert problemlos. Im folgenden Schritt soll *gNummer* ausgelesen werden. Dieser Versuch geht allerdings schief. Schaut man sich die Variable *test_gNummer* im Debugger an, so erhält man den Hinweis, das der Inhalt *undefined* ist also undefiniert ist. Somit wurde dasselbe erreicht wie in anderen OOP-Sprachen allerdings ohne das fehlende Schlüsselwort *private* zu verwenden.

Funktionen statt Methoden

Neben den Attributen sind die Methoden ein weiteres Merkmal von Klasse und Objekten. Die Bezeichnung Methode ist nur ein anderer Name für Funktion. Eine Funktion besteht aus einer oder mehreren Anweisungen. Sie kann am Ende ein Ergebnis zurückgeben, das muss aber nicht der Fall sein.

Eine Funktion wird über einen definierten Namen im Programm angesprochen. Es ist möglich, aber nicht notwendig, einer Funktion Argumente zur internen Verarbeitung zu übergeben. Die Argumente einer Funktion werden in der Regel nach dem Funktionsnamen definiert und auch übergeben. Die Argumente können Variablen sein, deshalb ist es wichtig, bei der Definition einer Funktion auf die korrekte Angabe des Typs zu achten. In JavaScript wird eine einfache-Funktion wie folgt deklariert:

```
function bspGetString() {
  return "Eine Zeichenkette";
}
```

Anschließend folgt der Funktionsname, über den diese aufgerufen werden kann. Es folgen eine öffnende und eine schließende Klammer. Zwischen den Klammern können Parameter platziert werden. Funktionen werden zwischen der ersten öffnenden Klammer { und der letzten schließenden Klammer } des übergeordneten Elements definiert:

```
(function () {
  function bspGetString()
  {
    return "Eine Zeichenkette";
  }
})();
```

Im Beispiel wurde eine Funktion *bspGetString* definiert, welche eine Zeichenkette als Wert zurück gibt.

Im letzten Abschnitt war bereits davon die Rede, dass einer Funktion Argumente übergeben werden können. Diese Argumente werden bei Funktionen auch als Parameter bezeichnet. Parameter sind immer dann nützlich, wenn in einer Funktion eine bestimmte Aufgabe erledigt werden soll, zu deren Lösung bestimmte Werte benötigt werden. Ein Beispiel: Eine Funktion *Addition* soll zwei Parameter (*p1* und *p2* ganzzahlige Variablen) entgegennehmen, diese addieren und das Ergebnis dann zurückgeben (Listing 1). Die Parameter werden innerhalb der Funktion wie lokal deklarierte Variablen benutzt. Die Definition der Parameter erfolgt innerhalb der Parameterliste direkt nach dem Funktionsnamen zwischen (und). Im Funktionskörper erfolgen die Addition der beiden Werte und die Rückgabe des Ergebnisses.

Listing 1: Parameter übergeben

```
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
    //Quellcode entfernt
    var result = Addition(5, 6);
  };
  function Addition(p1, p2) {
    return p1 + p2;
  }
})();
```

Eigenschaften

In der Regel soll es nicht möglich sein, auf Variablen einer Klasse direkt zugreifen zu können. Deshalb werden diese Variablen, wie bereits erwähnt, in anderen Sprachen auch oft mit dem Schlüsselwort *private* gekennzeichnet. Ein Zugriff von außen ist dann nicht mehr möglich. Aber natürlich muss auch auf den Inhalt solcher Variablen zugegriffen werden können. Ermöglicht wird dies durch die Verwendung von Eigenschaften (Properties). Auch JavaScript kennt Eigenschaften (Properties). Allerdings in einer etwas anderen Ausprägung als dies in anderen Programmiersprachen der Fall ist.

Alle Objekte in JavaScript unterstützen so genannte *expando Properties*. Es handelt sich hierbei um Eigenschaften, welche dynamisch zur Laufzeit angelegt werden. Einmal angenommen, man benötigt eine Eigenschaft Nachname in einem Objekt. Diese wurde aber dort nicht definiert. In JavaScript ist es nun möglich das Objekt einfach um diese Eigenschaft zu erweitern indem der Name der Eigenschaften an das Objekt angehängt wird obwohl er dort (eigentlich) nicht existiert. Dieser neuen Eigenschaft kann man dann Werte zuweisen und mit dieser arbeiten, als wäre diese bereits zuvor definiert worden. So ist in JavaScript das folgende Konstrukt problemlos umsetzbar (Listing 2).

Wie dem Listing gut zu entnehmen ist, sind innerhalb der Funktion *Person* keine Variablen oder Funktionen angelegt worden. Weder *Nachname*, *Vorname* noch *Personalnummer*. Trotzdem ist es kein Problem diese Variablen zur Laufzeit anzusprechen, obwohl es sie eigentlich nicht gibt. Den Eigenschaften *Nachname*, *Vorname* noch *Personalnummer* ▶

lassen sich Werte zuweisen und diese können auch wieder ausgelesen werden. Eine ausdrückliche Deklaration wie in anderen Sprachen ist somit nicht erforderlich. Diese Besonderheit/Komfort von JavaScript schafft aber auch Probleme. So lässt sich mit dieser Methode problemlos eine Variable anlegen (zum Beispiel durch einen Tippfehler) welche im restlichen Programm nicht vorhanden ist. Das kann die Fehlersuche natürlich erheblich erschweren. Deshalb sollte diese Vorgehensweise nur mit Bedacht verwendet werden.

Vielleicht ist es Ihnen aufgefallen? An einigen Stellen wird das Schlüsselwort *this* verwendet. Mit dem *this*-Schlüsselwort ist es möglich, innerhalb einer Klasse / einem Objekt auf die zugehörigen Bestandteile direkt zuzugreifen. Also einen direkten Bezug herzustellen. Auch JavaScript kennt das *this* Schlüsselwort. In folgendem Beispiel wird *this* verwendet, um die Variable *Kontostand* abzufragen (Listing 3).

In der Funktion *Konto* wurde die Variable *Kontostand* angelegt. Zur Abfrage der Variablen wurde die Funktion *getKontostand* implementiert. Innerhalb der Funktion wird die Variable *Kontostand* mittels des *this* Schlüsselwortes direkt referenziert.

Konstruktoren

Bei Anlage eines neuen Objekts wird automatisch eine mit dem Objekt verknüpfte Funktion aufgerufen. Diese Funktion trägt denselben Namen wie das Objekt und wird in dem Moment aufgerufen, in welchem das *new* Schlüsselwort zur Anwendung kommt (Listing 4). Um über den Konstruktor Variablen zur Initialisierung zu übergeben, müssen diese im Konstruktor benannt werden. Im Beispiel wird dem Konstruktor von *Konto* die Zahl 88888 übergeben. Die Übergabe findet innerhalb der Klammern nach dem Namen statt. Sobald die Zeile ausgeführt wird, wird innerhalb der Funktion *Konto* der dann in *_kontostand* enthaltene Wert der mit *this* referenzierten Variablen *Kontostand* zugewiesen.

Das Konzept der Klassen ist, wie bereits erwähnt, in JavaScript unbekannt. Da in der Regel nur Klassen voneinander erben können gibt es in JavaScript an dieser Stelle ein Problem. Eine Klasse wird in JavaScript deshalb als eine einfache Funktion mit dem Schlüsselwort *function* erstellt. Vererbung wird in JavaScript mit Hilfe der Prototypen-Funktion realisiert. Aber, was bedeutet das? Das folgende Vater / Sohn Beispiel soll das Ganze einmal verdeutlichen (Listing 5).

Was ist hier passiert? Es wurde eine Funktion *Vater* erstellt, welche eine Variable *Nachname* enthält. Diese Variable wurde mit dem Namen *Schmidt* initialisiert. Zusätzlich wurde die Funktion *getNachname* hinzugefügt mit welcher der Inhalt der Variablen abgefragt werden kann. Soweit so gut. Außerdem wurde ein weitere Funktion *Kind* welche über keinerlei Code verfügt geschrieben. Klar, das *Kind* soll alles vom *Vater* erben. Es ist aber keine Ableitung *Kind - Vater* erkennbar!?

Diese Ableitung wird erst zur Laufzeit mithilfe der Eigenschaft *prototype* realisiert. Mit *prototype* wird (wie es der Na-

Listing 2: Eigenschaften

```
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
    //Quellcode entfernt
    var person = new Person();
    person.Nachname = "Bleske";
    person.Vorname = "Christian";
    person.Personalnummer = 123456;
    var test_nachname = person.Nachname;
    var test_vorname = person.Vorname;
    var test_Personalnummer = person.Personalnummer;
};
})();
function Person() {
}
```

Listing 3: Das this-Schlüsselwort

```
JavaScript
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
    //Quellcode entfernt
    var mein_Konto = new Konto();
    var stand = mein_Konto.getKontostand();
  };
})();
function Konto() {
  this.Kontostand = 99999;
  this.getKontostand = function () {
    return this.Kontostand;
  }
}
```

Listing 4: Konstruktoren

```
var konto = new Konto();
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
    //Quellcode entfernt
    var mein_Konto = new Konto(88888);
    var stand = mein_Konto.getKontostand();
  };
  app.start();
})();
function Konto(_kontostand) {
    this.Kontostand = _kontostand;
    this.getKontostand = function () {
        return this.Kontostand;
    }
}
```

Listing 5: Vererbung

```
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
    //Quellcode entfernt
    Kind.prototype = new Vater();
    var sohn = new Kind();
    var test_nachname = sohn.getNachname();
  };
})();
function Vater() {
  this.Nachname = "Schmidt";
  this.getNachname = function () {
    return this.Nachname;
  }
}
function Kind() {
}
```

Listing 6: Überschreiben

```
(function () {
  //Quellcode entfernt
  app.onactivated = function (eventObject) {
    //Quellcode entfernt
    var n_Vater = new Vater();
    n_Vater.getName = function() {
      return "Hans Schmidt"
    };
    var test_name = n_Vater.getName();
  };
  app.start();
})();
function Vater() {
  this.Name = "Schmidt";
  this.getName = function () {
    return this.Name;
  }
}
```

me schon vermuten lässt) eine Vorlage für das aktuelle Objekt festgelegt. Diese Vorlage ist die *Vater*-Funktion. Nach dieser Zuweisung kann jetzt problemlos ein neues *Kind*-Objekt (*sohn*) erzeugt werden. Dieses Objekt verwendet die in *prototype* abgelegte Vorlage (Funktion *Vater*) als Vorlage für das neue Objekt und so erhält der *Sohn* den *Nachnamen* des *Vaters* beziehungsweise die Funktion *getNachname* kann abgerufen werden. An dieser Stelle wird der Unterschied von JavaScript zu anderen Programmiersprachen besonders deutlich: Während die Vererbung in in anderen Sprachen durch das Klassenkonzept realisiert wird, geschieht dies in JavaScript durch das verändern von Objekten.

In JavaScript ist es möglich, Funktionen und Eigenschaften zu überschreiben. Das bedeutet, eine Funktion die in einem Objekt vorhanden ist, kann in einem abgeleiteten Objekt unter demselben Namen erneut implementiert werden. In JavaScript ist es jederzeit möglich, Elemente zu überschreiben (Listing 6). Im Beispiel wird in der Funktion *Vater* innerhalb der Funktion *getName* lediglich der Nachname (Schmidt) definiert. Später im Programm muss neben dem Nachnamen aber zusätzlich noch der Vorname ausgegeben werden. Wie wird das gemacht? Der Funktion *getName* wird einfach ein neuer Wert zugewiesen. In diesem Fall eine neue Funktion. Die Funktion wird so neu geschrieben, dass dann die erweiterte Zeichenkette zurückgegeben wird.

JavaScript kennt weder das Konzept der Namespaces, bekannt aus C#, noch das der Packages aus Java. Also gibt es keine (implementierte) Möglichkeit den Quellcode vernünftig zu strukturieren? Doch, hier gibt es einen Workaround welchen man verwenden kann. Das folgende Listing zeigt ein Beispiel zur Strukturierung von Quellcode in JavaScript:

```
var MeineBibliothek = {
    A: function() {
    },
    B: function() {
    }
}
// Anwendung:
MeineBibliothek.A();
MeineBibliothek.B();
```

Das Konzept im Listing beruht (abermals) darauf die Funktionen in Variablen zusammenzufassen und so eine Strukturierung des Codes zu erreichen.

Fazit

Wie Sie sehen, kann auch in JavaScript objektorientiert programmiert werden (Bild 2). Die grundlegenden Bausteine zur Programmierung mit Objekten sind also auch in JavaScript vorhanden. ■

Link zum Thema

- Online-Entwicklungsumgebung für JavaScript
  https://jsfiddle.net

Christian Bleske
ist Autor, Trainer und Entwickler mit dem Schwerpunkt Client / Server und mobile Technologien. Erreichbar ist er unter **cb.2000@hotmail.de**