JS Crashkurs

Einige JavaScript spezifische Dinge zusammengefasst.

Datentypen

Javascript kennt folgende primitiven Datentypen:

- undefined
- number: Beispiele: 1 1.0 .0 1. 0x1 1e0
- boolean: true oder false
- string: Beispiele: “Test”, ‘test’, `Seitentitel: ${document.title}`
- array: Beispiele: [], [1,2,3]
- function: Beispiele: function(){}, ()=>{}

Alle anderen Datentypen sind vom typ *Object*. Primitive Datentypen können in ein Objekt verpakt werden:

```
Object(123); // Ergibt ein Number object
Object("Test"); // Ergibt ein String object
new String("Test"); // Ebenfals ein String object
(new String("Test")).valueOf(); // valueOf gibt in diesem
 fall kein String Objekt, sondern wieder einen string zur
ück.
```

Object und String sind Klassen. *valueOf* ist eine Funktion. In JavaScript sind Klassen = Funktionen. Alles hinter dem *//* sind

einzeilige Kommentare. Mehrzeilige Kommentare sind zwischen */* und */* Der einfachste Weg in JavaScript Objekte zu erzeugen ist folgendermassen:

```
var x = {
  key: "value",
  a: 123,
  func(){}
};
```

Grundlegende Sprachkonstrukte

Variablen & Scope

Variablen definiert man mit *var, let* oder *const*. Variablen mit *var* davor sind im momentanen Funktionsscope gültig, variablen mit *let* oder *const* im momentanen block scope. const variablen sind unveränderbar. Ein Scope ist ein Sichtbarkeitsbereich bzw. Gültigkeitsbereich in welchem die Variable existiert.

```
var x; // Globale variable x
let y; // Globale variable y
{ // Neuer Blockscope
  var z = 1; // Globale variable z mit wert 1
  let w = 1; // Lokale variable w mit wert 1
} // Ende des Blockscopes, z existiert noch, w nichtmehr
(function(){ // Function expression, neuer Funktionsscope
```

```
  var q; // locale variable z, existiert nur innerhalb de
r Funktion
  let r; // locale variable r, existiert nur innerhalb de
r Funktion
})(); // Funktion der Function expression aufrufen & ende
 des Funktionsscopes
```

Neben dem Globalen, Localen und Blockscope wird manchmal auch vom Closure-Scope geredet. Wenn eine Funktion in einer Funktion eine Variable der darüberliegenden Funktion nutzt, liegt diese in einem Closure-Scope.

Funktionen

Eine Funktion kann alleine stehen, eine Funktion expression sein, oder eine Arrow function sein. Letztere ist in anderen Programmiersprachen als lambda oder closure function bekannt.

Beispiel für Funktion:

```
function test(a,b,c){ // Funktion test mit Parametern a,
b, c und imlizitem parameter this.
}
```

Funktionen dürfen in Funktionen oder Global definiert werden, aber nicht in einem Block scope.

Beispiele function expression:

```
var test = function test(a,b,c){ // Funktion expression mit name test, Parametern a, b, c und imlizitem parameter this, wird in variable test gespeichert.
};
var test2 = function(a,b,c){ // Funktion expression mit Parametern a, b, c und imlizitem parameter this, wird in variable test2 gespeichert.
};
(function(a,b,c){ // Funktion expression mit Parametern a, b, c und imlizitem parameter this.
});
test(); // Aufruf der funktion, die durch die Variable test referenziert wird
```

Eine Function expression ist, wenn man es nicht mit einer alleinstehenden Funktion zutun hat, sondern mit einer die teil eines Ausdrucks, einer expression ist. In gesprächen unterscheidet man für gewöhnlich nicht zwischen Function expression und Function, aber beim Programmieren ist es relevant.

Beispiel für Arrow funktionen:

```
x => x // Arrow funktion mit argument x, welche argument x zurück gibt
(a,b,c) => a+b+c // Arrow funktion mit argumenten a,b & c
```

```
, welche das Ergebnis der expression a+b+c zurück gibt.
()=>{return 0;} // Arrow funktion ohne elemente, die 0 zu
rück gibt.
```

Arrow funktionen haben kein eigenes *this*.

Objekte, Objektzugriffe & Methodenaufrufe

Ein Objekt hat keys/members/properties, also benannte Eigenschaften, welchen ein Wert zugeordnet wird. Beispiel:

```
var x = {
  message: "Hello",
  print(){
    alert(this.message); // übergabe des members message
des Objekts referenziert durch die konstante this als par
ameter an die funktion alert
  }
}; // Speichere Objekt mit member message und methode/fun
ktion print in variable x.
x.print(); // Aufruf der Methode x.print mit this=x, Ausg
abe "Hello"
x.print.call({message:"World"}); // Aufruf der Methode x.
print this={message:"World"}, Ausgabe "World"
x["print"](); // Aufruf der Methode x.print mit this=x, A
usgabe "Hello"
```

```
x["print"]["call"]({message:"World"}); // Aufruf der Meth
ode x.print this={message:"World"}, Ausgabe "World"
```

Mit dem Punkt greift man auf Objektmember/property/eigenschaft zu. Alternativ gehen auch eckige Klammern. Das Objekt kann hierbei der Inhalt einer Variable, der Rückgabewert einer Funktion, oder das Ergebnis eines Ausdrucks sein, z.B. *(true).toString()*. Der versuch auf einen member von *undefined* oder *null* zuzugreifen produziert einen TypeError, da diese keine member haben. Der Aufruf einer Funktion über eine Eigenschaft eines Objekts führt dazu, das der implizite this parameter der Funktion auf das Objekt gesetzt wird, welches die eigenschaft hatte. Primitive typen werden dabei vorher in ein Objekt konvertiert:

```
alert(typeof true); // ausgabe "boolean"
Boolean.prototype.test = function(){alert(typeof this);}
(true).test(); // Ausgabe object, da this ein Boolean obj
ekt ist, und kein boolean prmitiv.
```

Mehrere Variablen können das selbe Objekt referenzieren/beinhalten. Es ist dann egal, über welche man darauf zugreift. Es ist dann nicht wie wenn man 2 Taschen mit identischen Bällen hätte, sondern eher als hätte man 2 Taschen zu einer mit 2 öffnungen zusammengenäht, die einen Ball beinhaltet.

Einige Methoden & Eigenschaften sind bei allen Objekten vorhanden, z.B. *valueOf*, *toString*, *constructor*, etc.