Entwicklung Web-basierter Anwendungen

Prof. Dr. Stefan Linus Zander

Einführung in JavaScript | Functions

Outline

The Special Role of Functions in JavaScript

Functions in JavaScript are "First-Class Citizens"

- ⇝ *"you can do with functions anything you can do with objects"*

Functions have a **special internal property** called `[[Call]]` that other objects don't have

- The `[[Call]]` internal property is unique to functions and indicates that the object can be executed
- This property is not accessible via code but rather defines the behaviour of code as it executes
- JavaScript defines multiple internal properties for objects (indicated by the `[[...]]` double bracket notation)

Because functions are objects in JavaScript, they behave differently than functions in other languages.
⇒ Understanding this behavior is central to a good understanding of JavaScript.

Function Delarations vs. Function Expressions

Declaration

```
let result = add(5, 5);

function add(num1, num2) {
  return num1 + num2;
}
```

- Function declaration are **hoisted** to the top of the context[1]
  - ie., the function name is known ahead of time
- Hence, functions can be accessed before they are defined

Expression

```
let add = function(num1, num2) {
  return num1 + num2;
}; // mind the semicolon ';' at the expression's end

let result = add(5, 5);
```

- Function expressions are anonymous function – ie., functions without name
- function expression can not be hoisted; they can only be referenced through the variable
- Assign a function value to the variable `add`

[1] The context is either the function, in which the declaration occurs or the global scope, i.e., the `window` object

Hoisting

Function declarations are hoisted to the top of the context in which they are defined.

```
// ================================================
// Function Declaration
// ================================================

// OK
var result = add(5, 5);

function add(num1, num2) {
  return num1 + num2;
}
```

```
// ================================================
// Function Expression
// ================================================

// error!
var result = add(5, 5);

var add = function (num1, num2) {
  return num1 + num2;
};
```

What you can do with Functions in JavaScript

Since JavaScript functions are first-class functions, they can be used in the same way as objects, ie., anything that works with objects also works with functions!

1. **Assign a function to a variable**
2. **Add to an object**
3. **Pass to other functions as arguments**
4. **Return them from functions**

This makes JavaScript functions incredibly powerful!

Assigning a Function to a Variable

```
// Example 1
function sayHi() {
  console.log("Hi!");
}

sayHi(); // outputs "Hi!"

let say_something = sayHi;

say_something();       // outputs "Hi!"
```

```
// Example 2
function addOne(num) {
  return num + 1;
}

let plusOne = addOne;

let result = plusOne(1); // outputs '2'
```

```
// Example 3: Handler
function init() {
    alert("Page has been fully loaded.");
}

window.onload = init;
```

Adding a Function to an Object

Methods

- A property value of type function makes the property a **method**.
- Methods are treated the same way as properties except for they can be executed (i.e., their value is calculated).

```
var person = {
  name: "Nicholas",
  sayName: function() {
    console.log(person.name);
  }
};

person.sayName();   // outputs "Nicholas"
```

Think about

What is the difference between `person.name` and `this.name` ?

The `this` Object

- Every scope in JS has a `this` object
- `this` represents the calling object for the function
- In the global scope, `this` represents the global object[1]

```
function sayNameForAll() {
  console.log(this.name);
}
let person1 = {
  name: "Nicholas",
  sayName: sayNameForAll
};
let person2 = {
  name: "Greg",
  sayName: sayNameForAll
};
let name = "Michael";
person1.sayName();  // outputs "Nicholas"
person2.sayName();  // outputs "Greg"
sayNameForAll();    // outputs "Michael"
```

[1] The global object is the web browser `window`

Passing Functions to other Functions as Arguments

```
// (a-c) Using Functions as arguments
function calc(a, b, f) {
    return f(a,b);
}

function add(x,y) {
    return x+y;
}

function sub(x,y) {
    return x-y;
}

calc(9,4,add);
calc(9,4,sub);
```

```
// Example 2
function Person(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.decode = function() {
        return this.firstname + " " + this.lastname;
    }
}

function lower() {
    return this.firstname.toLowerCase() + " " + this.lastname.toLowerCase();
}

function scribble() {
    var str = this.firstname + " " + this.lastname;
    var result = "";
    for (var i = 0; i < str.length; i++ ) {
        if (i % 2 == 0) {
            result = result + str.charAt(i).toUpperCase();
        } else {
            result = result + str.charAt(i).toLowerCase();
        }
    }
    return result;
}

function dashed() {
    var str = this.firstname + " " + this.lastname;
    var result = "";
    for (var i = 0; i < str.length; i++ ) {
        if (i != this.firstname.length) {
            result = result + "_";
        }
        else result = result + " ";
    }
    return result;
}

let p = new Person("Peter", "Pan");
p.decode();
p.decode = scribble;
p.decode();
```

Returning a Function from a Function

```
function saySomething(name) {
    return function sayHello() {
        console.log("Hello, my name is ... " + name);
        // 'this.name' does not work since it looks
        // for the value of a 'name' property in the
        // property to which saySomething was assigned to.
    };
}

var a = saySomething("Stefan"); // vs. var a = saySomething;

a(); // Outputs "Hello, my name is ... Stefan"
```

Fragen:

- Warum funktioniert `console.log("Hello, my name is ... " + this.name);` im obigen Beispiel nicht?
- Wie sähe der Funktionsaufruf aus, wenn stattdessen `let a = saySomething;` angegeben würde?

Parameters in Functions

JS allows to pass any numbers of parameters to any functions since function parameters are stored in an array-like data structure called `arguments`.

```
function sum() {
  "use strict";
  var result = 0,
      i = 0,
      len = arguments.length;

  while (i < len) {
      result += arguments[i];
      i++;
  }
  return result;
}

console.log(sum(1, 2));          // 3
console.log(sum(3, 4, 5, 6));    // 18
console.log(sum(50));            // 50
console.log(sum());              // 0
```

Function Overloading

Many OO-languages support function overloading (=a combination of function name plus the number and types of parameters the function expects). Since JavaScript functions can accept any numbers of parameters, they don't have signatures which means that function overloading is not possible.