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 Elements" ⇝ hence, anything that works with objects also works with functions!

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

⇒ This makes JavaScript functions incredibly powerful!

How does this work ?
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

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.

JavaScript defines multiple internal properties for objects (indicated by the `[[...]]` double bracket notation)

Function Delarations vs. Function Expressions

Declaration

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

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

- Declared functions are named functions
- Hoisted at the top of the scope (=context) in which they are defined
- Used for ...
  - normal named functions that return something
  - **constructor functions** to generate reference values

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
- Function expression can not be hoisted;
  - they can only be referenced through the variable
- Used for ...
  - assignment expressions
  - function parameters (e.g. for handlers)
  - methods
  - return values of functions

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
// ================================================

let result = add(5, 5);

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

✓ OK

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

let result = add(5, 5);

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

! Error

Function Types

Constructor Functions

- Functions invoked with `new` are called **Constructor Functions**
- Constructor functions ...
  - define reference types
  - return a newly created reference value
- The default return type is the new reference value (=instance of an reference type)
- The instantiation process can be manipulated by explicitly specifying the return value of a constructor function
- Constructor function should be named using the camel-case notation

```
function Person(firstname, lastname, birthyear) {
    "use strict";
    this._firstname = firstname;
    this._lastname = lastname;
    this.age = new Date().getFullYear() - birthyear;

    this.sayName = function() {
        console.log(
          "My name is " +
          this._firstname + " " +
          this._lastname);
    }
}

// Usage
let p = new Person("Hans", "Haas", 1919);
p.sayName(); // My name is Hans Haas
```

Self-Invoking Functions

- Function expressions can be made "self-invoking"
- aka Immediately Invoked Function Expression (IIFE)
- A self-invoking expression is started automatically without being called and will be removed immediately when finished
- IIFE create a local scope for variables and the function
  - i.e., they a not visible globally
  - no mismatch with identically named variables among scripts
- Function expressions will execute automatically if the expression is followed by `()`
- A function declaration cannot be self-invoked
- Parentheses `()` around the function indicate that it is a function expression

Example #1

```
(function () {
  let x = "Hello World";  // I will invoke myself
  console.log(x);
})();
```

Example #2

```
<!DOCTYPE html>
<html>
<body>
<p>Functions can be invoked automatically without being called</p>
<p id="demo"></p>
<script>
(function () {
  document.getElementById("demo").textContent =
    "Hello! I called myself";
})();
</script>
</body>
</html>
```

Please note: Anonymous functions are bound to a variable wherefore they are technically not anonymous; IIFE in contrast require more brackets but are technically indeed anonymous.
Source: https://www.mediaevent.de/javascript/self-executing-functions.html

Alternative Notations for Self-Invoking Functions

Three common notations exists for Self-Invoking Functions

```
// Crockford's preference - parens on the inside
(function() {
  console.log('Welcome to the Internet. Please follow me.');
} () );

//The OPs example, parentheses on the outside
(function() {
  console.log('Welcome to the Internet. Please follow me.');
}) ();

//Using the exclamation mark operator
//https://stackoverflow.com/a/5654929/1175496
!function() {
  console.log('Welcome to the Internet. Please follow me.');
}();
```

Spaces have been added for reasons of readability and comprehensibility.

Arrow Functions

- Arrow functions are a **short syntax** for function expressions
  - no `function` keyword
  - no `return` keyword & curly brackets `{}` in single statements
  - no `()` when only one parameter is expected
- Arrow functions do not have their own `this`
  - They use `this` from the calling context (ie. surrounding block)
  - They are not well suited for defining object methods
- Arrow functions are not hoisted
  - They must be defined before they are used
- There is no `arguments` object in arrow functions

Please note
Return and the curly brackets can only be omitted if the function is a single statement. Because of this, it might be a good habit to always keep them.

```
// ES5
var x = function(x, y) {
  return x * y;
}

// ES6
const x = (x, y) => x * y;
```

```
// Usual notation
let double = function (num) {
   return num * 2;
}

// Shortest possible notation
let double = num => num * 2;

// Recommended notation
let double = (num) => { return num * 2; }
```

Source: https://www.mediaevent.de/javascript/arrow-function.html

Working with Functions

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: Executable Property Values

- 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).

```
let 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 JavaScript has a `this` object
- `this` represents the calling object for the function
- In the global scope, `this` represents the global object

```
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"
```

The global object is the web browser `window`

Passing Functions to other Functions as Arguments

```
// 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;
```

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.
    };
}

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

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

```
function sum() {
  "use strict";
  let 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 (ie. 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 do not have signatures which means that function overloading is not possible.

Summary

Points to Remember

- Functions in javascript are objects that have an internal `[[call]]` property
  - Their values can be calculated
- Since functions are **first-class objects**, JavaScript allows to ...
  - assign functions to variables
  - add functions to objects
  - pass functions to other functions as arguments
  - return functions from functions
- Functions used with `new` makes them constructors
  - `new` creates a new object rather than copies are reference from the original object
  - The object to be created can be controlled using the `return` statement
- Functions can be defined as function declarations and function expressions
- The arrow-notation should be used with caution

Arrow-functions behave differently compared to function declarations (e.g. `this` refers to the global scope rather than to the current object.)