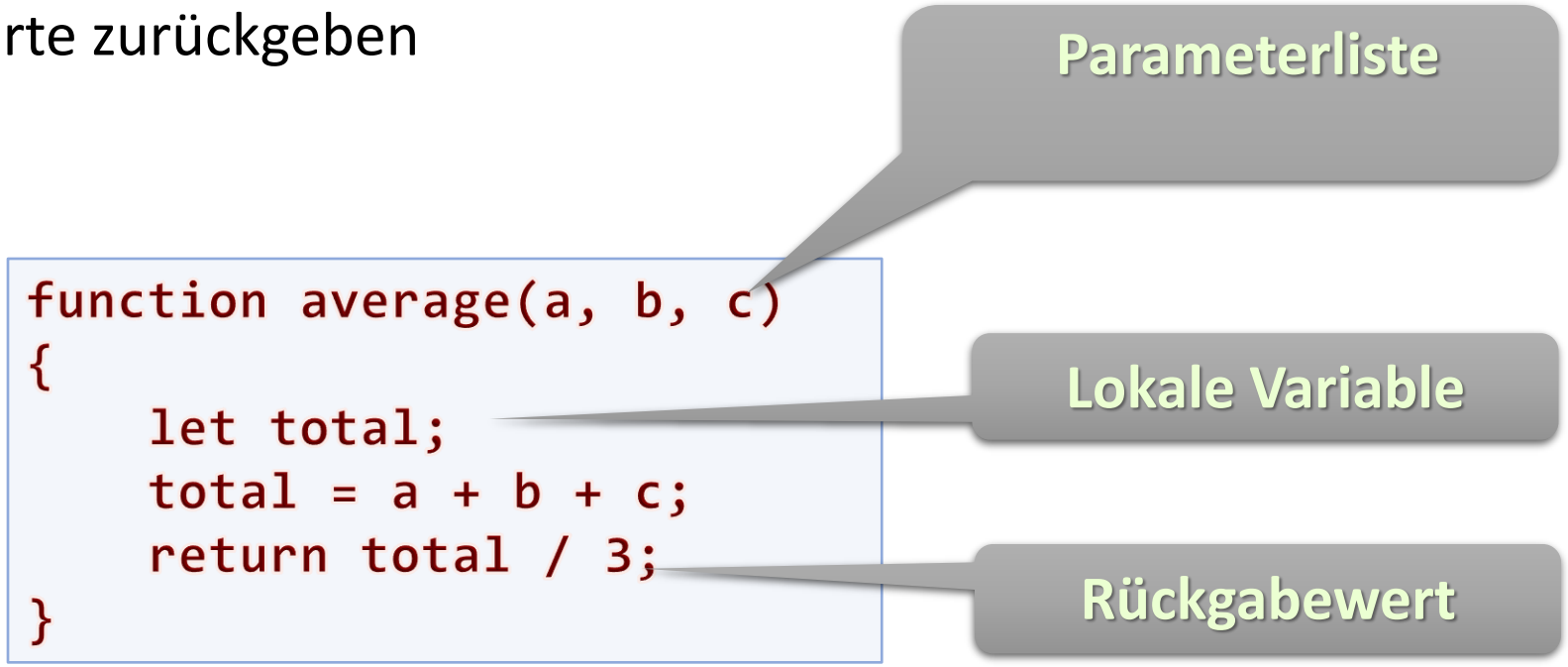
JavaScript – Funktionen

Ingo Köster

Diplom Informatiker (FH)

Funktionen

- Ähnlich zu anderen Sprachen:
  - Parameterlisten
  - Können Werte zurückgeben



Funktionen – Beispiel

```
> function sum(a,b)
  {
    let result = a + b;
    return result;
  }
```

```
> sum(3,4)
<· 7
> sum(3)
<· NaN
> sum(3,4,5,6)
<· 7
```

a = 3, b = 4

a = 3, b = undefined

a = 3, b = 4, restliche Parameter werden ignoriert

Parameter – Zusammenfassung

- Eine Funktion kann Parameter haben
- Parameter werden nur mit einem Namen, ohne Typ definiert
- Die Anzahl der übergebenen Argumente beim Aufruf sollte mit der Anzahl der Parameter übereinstimmen, muss es aber nicht!
  - Nicht zugewiesene Parameter sind „undefined“
  - Extra-Parameter werden „ignoriert“
    - Können jedoch verwendet werden

Variable Parameterlisten

› Wie in C/C++/C# können Funktionen auch Parameterlisten variabler Länge sinnvoll auswerten

› Dazu gibt es das Objekt **`arguments`** in jeder Funktion

› Enthält immer alle Übergabeparameter!

```
> function sum() {
    return arguments;
  }
  undefined
> sum(1,2,3,4,5,6);
  [1, 2, 3, 4, 5, 6]
>
```

Variable Parameterlisten – Beispiel

```
> function sum()
  {
      let s = 0;
      let len = arguments.length;
      for (let i = 0; i < len; i++)
      {
          s += arguments[i];
      }
      return s;
  }
```

```
sum(1,2,3,4,5,6,7,8,9,10);
55
```

Probleme mit `arguments`

- `arguments` ist jedoch kein Array, sondern ein Objekt, welches die Eigenschaft `length` hat und dessen Properties beginnend bei 0 indiziert sind
- Bedeutet, dass Funktionen von `Array` wie `forEach`, `map`, `reduce`, etc. nicht verwendet werden können
- Als Alternative kann der sog. Rest Operator verwendet werden
  - Ist immer der letzte Parameter einer Funktion!
  - **`...variablenName`**
- Weist `variablenName` alle verbleibenden Parameter einer Funktion zu

Beispiel Rest Operator

```
function sum(...numbers)
{
  let result = 0;
  for(let i = 0; i < numbers.length; i++)
  {
    result += numbers[i];
  }
  return result;
}
```

Weitere Beispiele

› Verwendung von `reduce` (mit `arguments` nicht möglich!)

```
function sum(...numbers) {
  return numbers.reduce((sum, next) => sum + next);
}
```

› Mehrere Parameter

```
function calc(operator, ...numbers) {
  console.log(operator);
  console.log(numbers);
}
```

› Aufruf

```
› calc('add', 1, 2, 3);
```

› **operator** enthält `'add'` und **numbers** das Array `[1, 2, 3]`

call by value und call by reference

- Alle Übergabeparameter in JavaScript werden per „call by value“ übergeben
- Für „call by reference“ gibt es keine Möglichkeit wie in anderen Sprachen Variablen ein & oder `ref` oder ähnliches voran zustellen
- „call by reference“ nur bei Übergabe von Objekten

Vertiefung

Sichtbarkeit von Variablen

- JavaScript unterscheidet zwischen drei Sichtbarkeitsebenen
- Gültigkeitsbereich in einem lokalen Block
  - Schlüsselwort `let`
  - Bevorzugt zu verwenden!
  - Wie in C#
- Gültigkeitsbereich in einer lokalen Funktion
  - Schlüsselwort `var`
- Globaler Gültigkeitsbereich
  - Ohne Angabe von `var` oder `let`
  - Einsatz sollte vermieden werden!
  - Im sog. Strict Mode nicht möglich

Vergleich zwischen var und let

Mit var

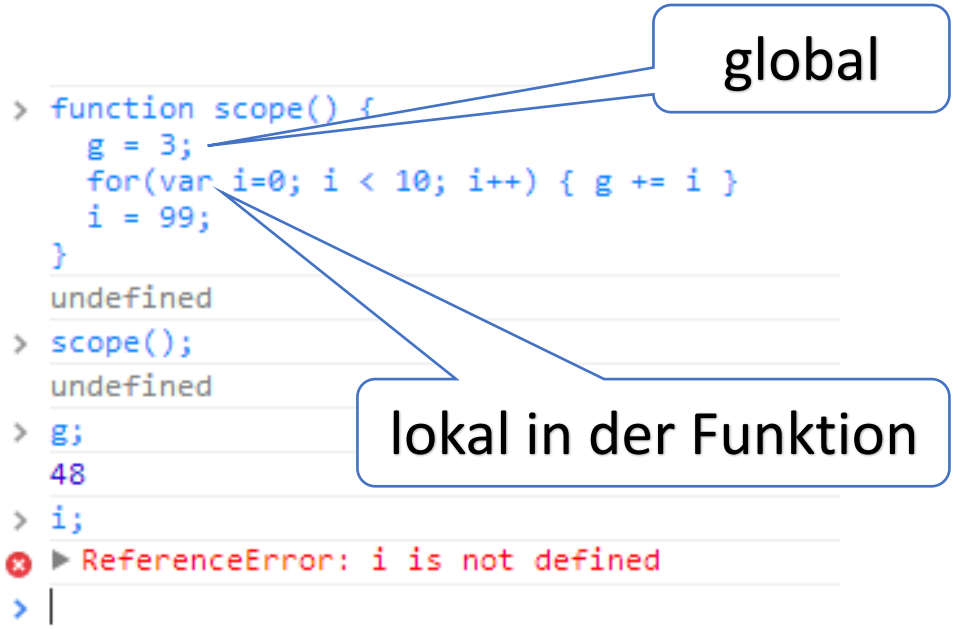
```
function varFunktion() {
  var x = 31;
  if (true) {
   // gleiche Variable!
   var x = 71;
   console.log(x);  // 71
  }
  console.log(x);  // 71
}
```

Mit let

```
function letTest() {
  let x = 31;
  if (true) {
   // andere Variable!
    let x = 71;
    console.log(x);  // 71
  }
  console.log(x);  // 31 !!!
}
```

Sichtbarkeit von Variablen

› Durch globale und funktionslokale Variablen ist JavaScript ist für Programmierer anderer Sprachen etwas ungewöhnlich



› Anders als in C# oder Java sind globale Variablen nicht nur in dem Block bzw. der Funktion „sichtbar“, in dem/der sie definiert wurden!

Sichtbarkeit von Variablen

- Werden Variablen ohne das Schlüsselwort `var` oder `let` angelegt, handelt es sich um globale Variablen
- Dabei spielt es keine Rolle ob diese innerhalb oder außerhalb einer Funktion angelegt wurden
- Solche Variablen sind auch über mehrere JavaScript-Dateien sichtbar!
  - Auf den Einsatz von globalen Variablen sollte verzichtet werden!

Anonyme Funktionen

› Funktionen sind Daten, können daher auch in Variablen gespeichert werden!

› Solche Funktionen haben keinen Namen!

```
let f = function()
{
    return 1;
}
```

```
f();
1
```

```
typeof f
"function"
```

Funktionen sind Daten!!!

› Funktionen sind in JavaScript echte Datentypen

› Wie in C#, können Funktionen u.a. als Argumente dienen

```
function aggregate(arr, start, func) {
  let result = start;
  for (let item in arr) {
    result = func(result, arr[item]);
  }
  return result;
}
let sum = function(arr) {
  return aggregate(arr, 0,
    function(a, b) { return a + b; } );
};
alert(sum([1, 2, 3]));
```

Funktionen sind Daten - Beispiele

```
function aggregate(arr, start, func) {
  let result = start;
  for (let item in arr) {
    result = func(result, arr[item]);
  }
  return result;
}
```

```
let sum = aggregate([1,2,3,4,5], 0, function(a,b) { return a + b;});
```

```
sum
```

```
15
```

```
let prod = aggregate([1,2,3,4,5], 1, function(a,b) { return a * b;});
```

```
prod
```

```
120
```

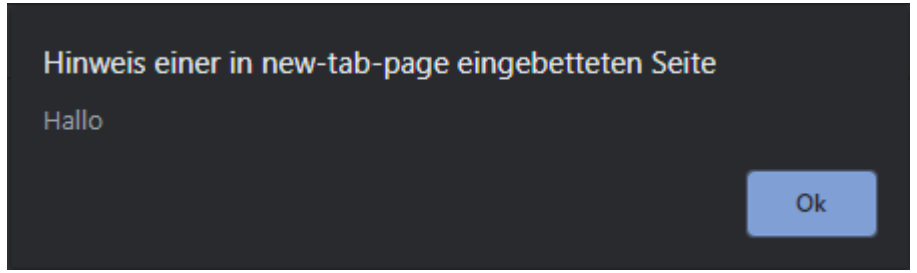
Direkte Funktionsausdrücke

- Direkte Funktionsausdrücke können dazu verwendet werden den eigenen Code besser zu strukturieren
  - Self Invoking Functions oder Immediate Functions

- Es wird eine Anonyme Funktion erstellt

- Zum Schluss ruft sich diese Funktion mittels des ( ) selbst auf

```
(function (param) {
    alert(param);
})("Hallo");
```



Lambda-Funktionen - Arrow Functions

- Seit ECMAScript 6 können anonyme Funktionen über Lambda-Ausdrücke definiert werden
  - Werden in JavaScript auch Arrow Functions genannt

- Aufbau und Verwendung haben sehr große Ähnlichkeit zu C#

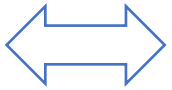
Lambda Beispiel

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let even = numbers.filter( e => e % 2 == 0 );
// 2, 4, 6, 8, 10

let odd = numbers.filter( e => e % 2 != 0 );
// 1, 3, 5, 7, 9
```

```
e => e % 2 != 0
```



```
function (e) { return e % 2 != 0; }
```

Lambda Beispiel

```
let words = ["one", "two", "three", "four", "five"];
let three_letters_long = [];

words.forEach(e =>
{
   if(e.length == 3)
   {
      three_letters_long.push(e);
   }
});

// three_letters_long == ["one", "two"]
```

Lambda Beispiel

```
function aggregate(arr, start, func)
{
    let result = start;
    for (let item in arr)
    {
        result = func(result, arr[item]);
    }
    return result;
}

let sum = aggregate([1, 2, 3, 4, 5], 0, function (a, b) { return a + b; });

let prod = aggregate([1, 2, 3, 4, 5], 1, (a, b) => a * b);
```