Web-Entwicklung

JavaScript

Inhalte der Vorlesung

- JavaScript
  - Kommentare
  - Variablen, Konstanten und Datentypen
  - Operatoren
  - Funktionen
  - Kontrollstrukturen
  - Fehlerbehandlung
  - Weitere Datenstrukturen
  - Promises und async/await

Kommentare

```
// single line comment


/*
multi
line
comment
*/
```

Variablen, Konstanten und Datentypen

Dynamische Typisierung

- Statische Typisierung (z.B. in Java oder C#): Datentypen aller Ausdrücke sind bereits zur Compile-Zeit bekannt
  - Überprüfungen im Zusammenhang mit dem Datentyp können bereits zur Compile-Zeit durchgeführt werden, z.B. Existenz von Methoden, die aufgerufen werden sollen
- Dynamische Typisierung (z.B. in JavaScript): Datentypen von Ausdrücken sind erst zur Laufzeit bekannt
  - Werte haben einen Typ, aber nicht Variablen/Konstanten!
  - Eine Variable kann im Verlauf auf Werte unterschiedlichen Typs zeigen

Variablen (var/let)

- Variablendeklaration mithilfe des Schlüsselworts `var` bzw. `let`
- Variablenbezeichner
  - beginnen mit einem Buchstaben, Unterstrich oder Dollar-Zeichen, anschließend folgen Buchstaben, Ziffern und Unterstriche
  - sind case-sensitive
- Globale Variablen
  - Lässt man das Schlüsselwort weg, wird die Variable zu einer globalen Variable
  - Eigenschaft des sog. *global object*
  - Überschreibt ggf. bereits vorhandene, globale Variable!

```
var value = 'abc‘;

let value2 = 42;
value2 = 'Forty-two';       // 👎

globalVar = '...';          // 👎
```

Variablen (var/let)

- Die Gültigkeit einer Variablen, die mit …
  - … `var` deklariert wurde, wird durch Funktionen begrenzt (*function level scope*)
    - Innerhalb einer Funktion definierte Variablen sind in der gesamten Funktion gültig
      - Auch in inneren Anweisungsblöcken definierte Variablen
      - Auch vor der Deklaration
        - *Hoisting*: Implizites "Anheben" der Deklaration aller Variablen an den Beginn des Funktionsrumpfes
  - … `let` deklariert wurde, wird durch Blöcke begrenzt (*block level scope*)
    - z.B. nur innerhalb eines Schleifenrumpfes
    - Erlaubt auch mehrfache Deklaration der gleichen Variable innerhalb einer Funktion (Überdeckung)
      - Sollte vermieden werden!
- Empfehlung: Weitestgehender Verzicht auf `var` zugunsten von `let`

Variablen (let)

```
let i = 4711;

for (let i = 1; i <= 10; i++) {   // 👎
   console.log(i);   // 1 .. 10
}

console.log(i);      // 4711
```

Gültigkeitsbereiche

```
function testScope() {
   if (true) {
      var k = 1337;
   }
   for (var l = 0; l < 1024; l++) {
      // do something
   }
   for (let m = 0; m < 1024; m++) {
      // do something
   }
   console.log(k);                // 1337
   console.log(l);                // 1024
   console.log(m);                // ReferenceError
}
```

Gültigkeitsbereiche

```
function testScope() {
   …
   console.log(i);   // undefined, no ReferenceError
   var i = 5503;     // Hoisting
   console.log(i);   // 5503
}
```

Gültigkeitsbereiche

```
function testScope() {
   …
   var n = 2048;
   function innerFunction () {
      console.log(n);
      var innerVar = true;
   }
   innerFunction();       // 2048
   console.log(innerVar); // ReferenceError
}
```

Konstanten (const)

- Schlüsselwort `const` erlaubt die Deklaration von Konstanten
  - Deklaration und Initialisierung muss in einer Anweisung erfolgen
  - Anschließende Zuweisungen an den Bezeichner erzeugen Laufzeitfehler
    - Verhindert nicht die Änderung von Objekteigenschaften
  - *block level scope* (wie `let`)

```
const pi = 3.141;
pi *= 2;  // TypeError: Assignment to
          // constant variable.

const obj = { value: 2 };
obj.value = 3;     // no(!) TypeError
console.log(obj);  // { value: 3 }
```

Datentypen

- Primitive Datentypen
  - `Boolean`
  - `Number`
  - `String`
- Spezielle primitive Datentypen
  - `Undefined`
  - `Null`
- Referenztypen
  - `Object` (Assoziative Arrays)
    - `Array` (Arrays)
    - `Function` (Funktionen)
    - …

typeof

- `typeof`-Operator liefert Datentyp des Operanden als String
- Weitergehende Möglichkeiten zur Typüberprüfung (`instanceof`) in nächster Vorlesungseinheit

Typ	**Rückgabewert**
Boolean	`'boolean'`
Number	`'number'`
String	`'string'`
Undefined	`'undefined'`
Null	`'object' (!)`
Function	`'function'`
Alle anderen Objekte	`'object'`

Boolean

- Wahrheitswerte `false/true`
- Alle Ausdrücke können vom Interpreter zu einem Wahrheitswert evaluiert werden („falsy/truthy“)
  - `false`
    - Leere Strings
    - `0`
    - `null`
    - `undefined`
    - `NaN`
  - `true`: Alle anderen Ausdrücke
  - Erlaubt häufig stark verkürzte Bedingungen

Number

- Nur ein Datentyp für ganze und nicht-ganze Zahlen
  - Statische Methode `Number.isInteger()`
- `Number` repräsentiert eine 64 Bit-Gleitkommazahl gemäß IEEE 754
  - Entspricht dem Datentyp `double` in Java
  - Kein Datentyp für 32 Bit-Fließkommazahlen (`float`)
- Wertebereich zwischen `Number.MIN_VALUE` und `Number.MAX_VALUE`

```
const n = 1024;
console.log(typeof n,
      Number.isInteger(n));   // number true

const r = 3.141;
console.log(typeof r,
      Number.isInteger(r));   // number false

const hex = 0xff;
console.log(hex);             // ?
```

Number

- Besondere Werte:
  - `Infinity`

    Repräsentiert die Unendlichkeit, z.B. bei Übertreten des Wertebereichs oder bei Division durch 0
  - `NaN`

    Für andere ungültige Zahlenwerte nimmt die Variable den Wert `NaN` (für *Not a Number*) an
- Globale Methoden zur Prüfung
  - `isFinite()`
  - `isNaN()`

```
const x = Number.MAX_VALUE * 2;
console.log(x);               // Infinity

const y = 1 / 0;
console.log(y);               // Infinity
console.log(isFinite(y));     // false

const z = Math.sqrt(-1);
console.log(z);               // NaN
console.log(isNaN(z));        // true
```

Number

- Globale Funktionen zur Umwandlung von Zeichenketten in `Number`-Objekte
  - `parseInt(string, radix)`
  - `parseFloat(string)`
- Funktionen parsen maximal-langen, gültigen Präfix
  - Teil-Zeichenkette ab erstem ungültigen Zeichen werden ignoriert
  - Whitespace-Präfix wird ignoriert

```
parseInt(' 0xF', 16);
parseInt('F', 16);
parseInt('17', 8);
parseInt('015', 10)
parseInt(15.99, 10);
parseInt('15,123', 10);

parseFloat('3.14');
parseFloat('314e-2');
parseFloat('0.0314E+2');
parseFloat('3.14more non-digit characters');
```

String

- UTF-16-kodierte Unicode-Zeichenketten
  - `length` liefert Anzahl Bytes zurück!
- Unveränderlich (*immutable*)
- Einfache oder doppelte Anführungszeichen
  - Erlaubt Verwendung des jeweils anderen Zeichens ohne Escape-Sequenz innerhalb der Zeichenkette
- Kein spezieller Datentyp für Einzelzeichen
  - `at()`, `charAt()` bzw. `[]`-Operator liefern String-Objekt
- Methoden: https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String

```
const tamil = 'This is தமிழ்';

const greeting = 'I ❤️ "JavaScript"';

console.log(typeof tamil.at(0));  // string

console.log('Hello'.length);   // 5
console.log(tamil.length);     // 13!
```

Template Strings

- Häufig müssen Werte von Variablen oder Rückgaben von Funktionen in eine Zeichenkette eingefügt werden
- Naiver Ansatz: String-Konkatenation
  - +-Operator
- Template Strings
  - Geklammert in Gravis (engl.: *back ticks*)
  - Variablen, Konstanten oder Funktionsaufrufe in `${}` geklammert

```
const name = 'Heinz';
const age = 42;

function getRandomName() {
   const names = ['Jane', 'John', 'Max'];
   return names[Math.floor(Math.random()
         * names.length)];
}

const text = `Hello ${getRandomName()}. My
      name is ${name} and I'm ${age} years
      old.`;

console.log(text);
```

Undefined - Null

- Datentypen mit jeweils nur einem möglichen Wert
- Beide Werte drücken aus, dass eine Variable nicht belegt ist
- `undefined`
  - Nicht initialisierte Variablen
  - Nicht existente Objekteigenschaften
  - Nicht übergebene Funktionsargumente
  - Rückgabe von Funktionsaufrufen ohne Rückgabewert
- `null`
  - Beabsichtigte Nichtbelegung einer Variablen durch den Entwickler

```
let u;
console.log(u, typeof u);   // undefined undefined

const v = 500;
console.log(v.nonExistentProperty, typeof
   v.nonExistentProperty);  // undefined undefined


const w = null;
console.log(w, typeof w);   // null object (!)
```

Arrays

- Container-Datentyp für Objekte beliebigen Typs
- Numerischer, 0-basierter Schlüssel
- Erzeugung über Literal-Kurzschreibweise oder sog. Konstruktorfunktion
  - Ein `Number`-Argument: Array-Länge, Elemente: `undefined`
  - Mehrere Argumente: Array-Inhalte
- Lesender und schreibender Zugriff über Index-Operator

```
const names = ['Tim', 'Struppi'];

const names2 = new Array();
names2[0] = 'Max';
names2[1] = 'Moritz';
names2[2] = 0;

const tenItems = new Array(10);
const threeItems = new Array(1, 2, 3);
```

Arrays

- Methode `at()` zum lesenden Zugriff auf Elemente indizierbarer Objekte
  - `String`
  - `Array`
- Vergleichbar mit `[]`-Operator
- Negative Indizes ermöglichen Auswahl relativ zum Ende des Objekts

```
const digits = '12345';

console.log(digits.at(1), digits.at(-1));  // 2 5

const chars = ['a', 'b', 'c', 'd', 'e'];

console.log(chars.at(1), chars.at(-1));    // b e
```

Arrays

- Weitere gängige Methoden
  - `concat()`
  - `filter()`
  - `join()`
  - `map()`
  - `pop()`
  - `push()`
  - `reverse()`
  - `shift()`
  - `slice()`
  - `splice()`
  - `sort()`
  - …
- https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array

Objekte – Assoziative Arrays

- Objekte in JavaScript sind assoziative Arrays, d.h. Container für Schlüssel-Wert-Paare
  - Schlüssel sind im Gegensatz zum Array nicht numerisch-aufsteigend, sondern alphanumerisch
  - Werte sind beliebige Objekte
- Ein Schlüssel-Wert-Paar nennt man auch
  - Eigenschaft
  - Methode, falls Wert eine Funktion ist
- Erzeugung von Objekten mithilfe des sog. Objekt-Literals
  - Weitere Erzeugungsmöglichkeiten in nächster Vorlesungseinheit
- Lesender und schreibender Zugriff auf Eigenschaften oder Methoden über Punkt- oder Index-Operator

Objekt-Literal

```
const phonebookEntry = {
   firstName: 'Hans',
   lastName: 'Wurst',
   number: '0123-456789'
};

console.log(phonebookEntry.number);           // 0123-456789
console.log(phonebookEntry['number']);        // 0123-456789


phonebookEntry.firstName = 'John';            // { firstName: 'John',
phonebookEntry['lastName'] = 'Doe';           //   lastName: 'Doe',
                                              //   number: '0123-456789' }
console.log(phonebookEntry);
```

Objekt-Literal

```
const phonebookEntry2 = {
   firstName: 'Hans',
   lastName: 'Wurst',
   number: '0123-456789',
   print: function() {
      console.log('%s, %s: %s‘, this.firstName, this.lastName, this.number);
   }
};

phonebookEntry2.print();   // Hans, Wurst: 0123-456789
```

Enumerationen

- Assoziative Arrays können auch zur Emulation von Enumerationen genutzt werden

```
const Color = {
   RED: 0,
   GREEN: 1,
   BLUE: 2
};
console.log(Color.RED);

const color = Color.BLUE;
console.log(color === Color.GREEN);
```

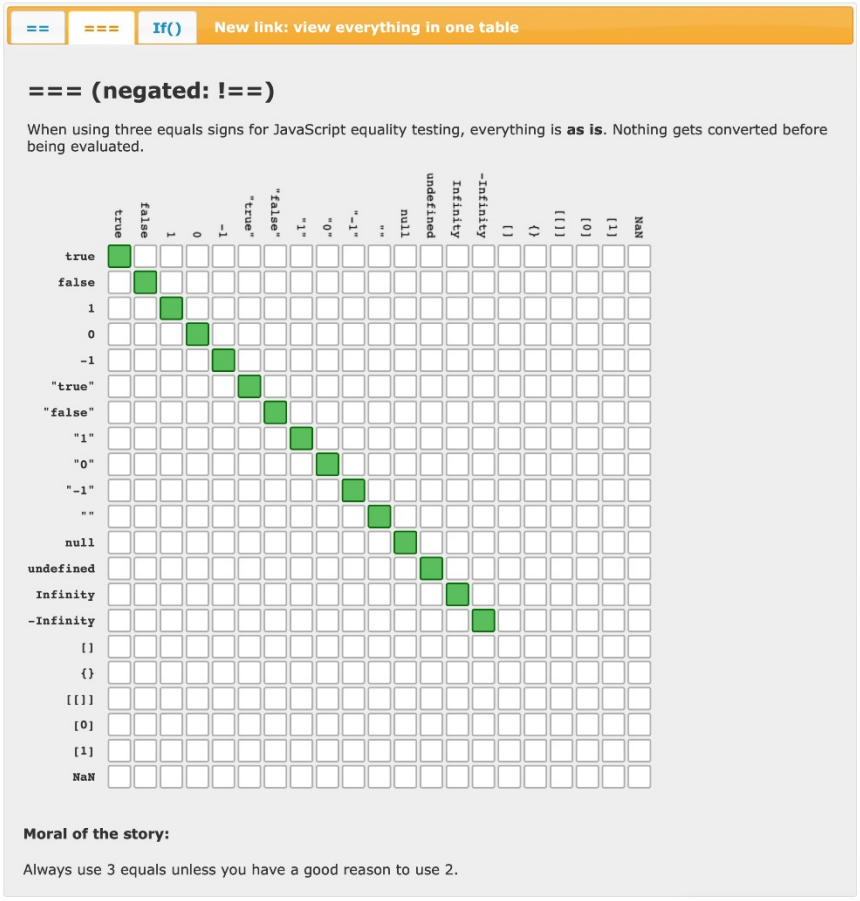
```
const Color2 = {
   RED: 'RED',
   GREEN: 'GREEN',
   BLUE: 'BLUE'
};
console.log(Color2.RED);

const Color3 = {
   RED: 0xFF0000,
   GREEN: 0x00FF00,
   BLUE: 0x0000FF
};
console.log(Color3.RED);
```

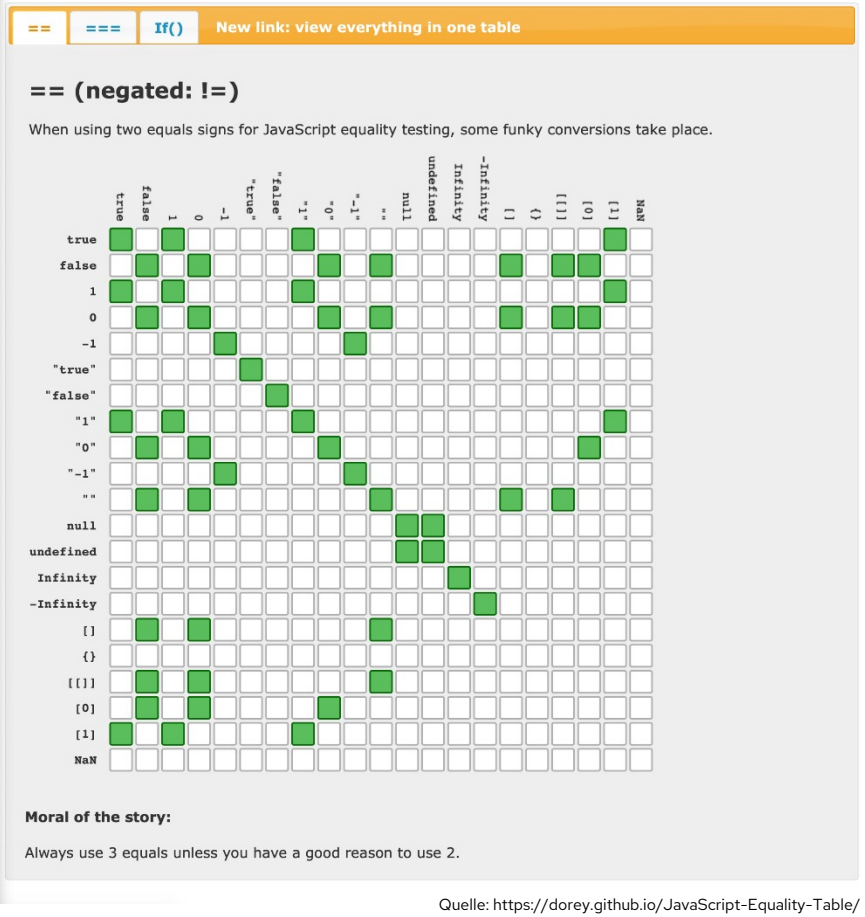
Operatoren

Vergleichsoperatoren

Operation	Operator	Beschreibung
Strikte Gleichheit	===	true, falls Operanden gleich sind und Datentypen übereinstimmen (weil eine implizite Typumwandlung nicht erlaubt ist)
Strikte Ungleichheit	!==	
Gleichheit	==	true, falls Operanden gleich sind (Implizite Typumwandlung erlaubt)
Ungleichheit	!=	
Größer	>	
Größer oder Gleich	>=	
Kleiner	<	
Kleiner oder Gleich	<=	

Vergleichsoperatoren





Quelle: https://dorey.github.io/JavaScript-Equality-Table/

Arithmetische Operatoren

Operation	Operator	Beschreibung
Modulo	%	Rest der ganzzahligen Division der Operanden
Inkrement	++ (Präfix oder Postfix)	Erhöht den Wert des Operanden um 1
Dekrement	-- (Präfix oder Postfix)	Senkt den Wert des Operanden um 1

Logische Operatoren

Operation	Operator	Beschreibung
Log. UND	&&	Liefert ersten Operanden zurück, falls dieser zu false evaluiert wird, sonst den zweiten Operanden
Log. ODER	||	Liefert ersten Operanden zurück, falls dieser zu true evaluiert wird, sonst den zweiten Operanden
Log. NICHT	!	Negiert den zu einen Wahrheitswert evaluierten Operanden

Bitweise Operatoren

Operation	Operator
Bitw. UND	&
Bitw. ODER	|
Bitw. XOR	^
Bitw. NICHT	~
Bitw. Linksverschiebung	<<
Bitw. Rechtsverschiebung	>>
Bitw. Rechtsverschiebung ohne Berücksichtigung des Vorzeichens	>>>

Weitere Operatoren

Operation	Operator	Beschreibung
Ternärer Bedingungsoperator	`<Bedingung>?<Wert1>:<Wert2>`	Liefert <Wert1>, falls Bedingung zu true evaluiert wird, sonst <Wert2>
Optional Chaining-Operator	`<Objekt>?.<Eigenschaft>`	Liefert undefined, falls <Objekt>.<Eigenschaft> null oder undefined, sonst <Objekt>.<Eigenschaft>
Nullish Coalescing-Operator	`<Wert1> ?? <Wert2>`	Liefert <Wert2>, falls <Wert1> null oder undefined, sonst <Wert1>
Löschoperator	`delete`	Löscht Objekt-eigenschaften oder Array-Elemente
Eigenschaftsexistenzoperator	`<Eigenschaft> in <Objekt>`	true, falls <Objekt> die <Eigenschaft> besitzt (traversiert Prototypenkette)
Prototypüberprüfung	`<Objekt> instanceof <Typ>`	true, falls <Typ> in der Prototypenkette von <Objekt> enthalten ist
Typbestimmung	`typeof <Operand>`	Liefert Typ des Operanden (String)

Ternärer Bedingungsoperator (?:)

- Erlaubt vereinfachte Rückgabe eines zweier Werte in Abhängigkeit von einer Bedingung
  - Häufig kompakter als `if`-Verzweigung
- Ist der erste Operand `true`, liefert der gesamte Ausdruck den zweiten Operand zurück, ansonsten den dritten Operand

```
const p1 = {
   firstName: 'Hans', lastName: 'Wurst',
   car: { manufacturer: 'ACME', model: 'Pinky' }
};

console.log(`${p1.firstName} ${p1.lastName}
      ${p1.car ? ': ' : ''}
      ${p1.car ? `${p1.car.manufacturer} ${p1.car.model}` : ''}
`);
```

Optional Chaining (?.)

- Erlaubt vereinfachten Zugriff auf Eigenschaften geschachtelter Objekte
- Ist der linke Operand `null` oder `undefined` ("nullish"), liefert der gesamte Ausdruck den Wert `undefined`
- Anderenfalls wird die Eigenschaft rechts des Operators dereferenziert

```
const p1 = {
   firstName: 'Hans', lastName: 'Wurst',
   car: { manufacturer: 'ACME', model: 'Pinky' }
};

const p2 = {
   firstName: 'Max', lastName: 'Mustermann'
};

console.log('%s %s',          // ACME Pinky
   p1.car.manufacturer, p1.car.model);
console.log('%s %s',          // TypeError
   p2.car.manufacturer, p2.car.model);
console.log('%s %s',          // undefined undefined
   p2?.car?.manufacturer, p2?.car?.model);
```

Optional Chaining (?.)

- `?.` kann auch zwischen Methodenname und Aufruf-Klammerpaar eingesetzt werden
- Methodenaufruf liefert dann `undefined`, falls Methode nicht existiert

```
const a = {
  b: {
    c: {
      d: {
        m: function () {
         return 'Hello';
        }
      }
    }
  }
};

console.log(a && a.b && a.b.c && a.b.c.d
    && a.b.c.d.m && a.b.c.d.m());

console.log(a?.b?.c?.d?.m?.());
```

Nullish Coalescing (??)

- Erlaubt Angabe von Fallback-Werten für Ausdrücke
- Ist der linke Operand nicht `null` oder `undefined` ("nullish"), wird dieser Ausdruck zurück geliefert
- Anderenfalls wird der rechte Operand zurück geliefert

```
const p1 = {
  firstName: 'Hans', lastName: 'Wurst',
  salary: 55000
};

const p2 = {
  firstName: 'Max', lastName: 'Mustermann'
};

console.log(p1.salary ?? 48000);   // 55000
console.log(p2.salary ?? 48000);   // 48000
```

Nullish Coalescing (??)

- Ähnelt dem logischen ODER-Operator (`||`)
- Dieser liefert aber nicht nur bei "nullish"-Werten den rechten Operand, sondern bei allen "falsy" Werten
  - Leere Strings
  - `0`
  - `null`
  - `undefined`
  - `NaN`

```
const p2 = {
  firstName: 'Max', lastName: 'Mustermann'
};

const p3 = {
  firstName: 'Poor', lastName: 'Boy',
  salary: 0
};

console.log(p2.salary || 48000);  // 48000
console.log(p3.salary || 48000);  // 48000 (!)
```

Funktionen

Funktionsanweisung

- Deklaration von Funktionen mithilfe des Schlüsselworts `function`
- Keine Typangaben bei Argumenten oder Rückgabe
  - Keine Typsicherheit

```
function add(arg1, arg2) {
   return arg1 + arg2;
}

console.log(add(1, 2));                    // 3
console.log(add('Hello', 'World'));        // HelloWorld
```

Funktionsanweisung

- Typsicherheit kann erst zur Laufzeit hergestellt werden

```
function add2(arg1, arg2) {
   if (typeof(arg1) !== 'number' || typeof(arg2) !== 'number') {
      throw new TypeError('Arguments of "add" must be of type "number"');
   }
   return arg1 + arg2;
}

console.log(add2('Hello', 'World'));                // TypeError
```

Funktionsausdruck

- Alternative: Erstellung einer (anonymen) Funktion und Zuweisung zu einer Variablen
- Im Gegensatz zu Funktionsanweisungen stehen solche Funktion erst nach der Zuweisung zur Verfügung

```
const add3 = function (arg1, arg2) {
   return arg1 + arg2;
};

console.log(add3(1, 2));          // 3
```

Function-Objekte

- Funktionen werden durch Objekte repräsentiert
- Eigenschaften
  - `name`
  - `length`
  - …
- Funktionsanweisung erzeugt Instanz und gleichnamige Variable

```
function add(arg1, arg2) {
   return arg1 + arg2;
}
```

Funktionen als First Class Objects

- Funktionen sind First Class Objects, d.h. sie können wie Objekte primitiver Datentypen
  - einer Variablen zugewiesen werden

```
function add(arg1, arg2) {
   return arg1 + arg2;
}

const plus = add;
console.log(add(1, 2));
console.log(plus(1, 2));
```

Funktionen als First Class Objects

- Funktionen sind First Class Objects, d.h. sie können wie Objekte primitiver Datentypen
  - als Rückgabe einer Funktionen dienen
- Erstellung und Rückgabe einer anonymen Funktion

```
function createPow(exponent) {
   return function (x) {
      let result = x;
      for (let i = 1; i < exponent; i++) {
         result *= x;
      }
      return result;
   };
}

console.log(createPow(10)(2));   // ?
console.log(createPow(3).name);  // ?
```

Funktionen als First Class Objects

- Funktionen sind First Class Objects, d.h. sie können wie Objekte primitiver Datentypen
  - als Argument einer Funktionen dienen
- z.B. in Array-Methoden
  - `forEach()`
  - `map()`
  - `filter()`
  - `every()`
  - `some()`
  - `reduce()`
  - …
- Später: Callback-Entwurfsmuster, Events

```
function isEven(element, index, array) {
   return element % 2 === 0;
}

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

console.log(numbers.filter(isEven));
```

Funktionsargumente

- Beim Aufruf von Funktionen kann die Anzahl der aktuellen Argumente von der Anzahl der formalen Parameter abweichen
- Implizites Array-ähnliches Objekt `arguments` enthält innerhalb von Funktionen alle übergebenen Argumente
  - `length`-Eigenschaft
  - Index-Operator
  - Aber nicht alle Array-Funktionen
- Problematisch: Funktionssignatur gibt keinen Hinweis darauf, ob die Funktion kein Argument oder beliebig viele Argumente erwartet
  - Besser: Rest-Parameter

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

console.log(sum(1, 2, 3));       // 6
console.log(sum(1, 2, 3, 4, 5)); // 15
```

Optionale Funktionsparameter

- Überladen von Funktionen ist nicht möglich
- Realisierung optionaler Parameter möglich durch
  - Optionsobjekt
  - Standardwerte für Funktionsparameter

Optionsobjekt

- Zusammenfassung ggf. vorhandener optionaler Parameter in einem assoziativen Array
- Übergabe des Arrays als letztes, aber einziges optionales Argument an die Funktion
  - Nur benötigte Parameter werden explizit gesetzt
- Abfrage z.B. via Optional Chaining-Operator
- Problem: Funktionssignatur enthält keinen Hinweis auf konkrete optionale Parameter innerhalb des Optionsobjekts

```
function add4(a, b, config) {
   const result = a + b;
   if (config?.log) {
      console.log(result);
   }
   return result;
}

add4(5, 2, { log: true });  // 7
add4(5, 2, { log: false }); //
add4(5, 2);                 //
```

Standardwerte für Funktionsparameter

- Angabe von Standardwerten für Funktionsparameter möglich
- Wertzuweisung oder Funktionsaufruf in Parameterliste
- Nach Parameter mit Standardwert darf kein Parameter ohne Standardwert folgen

Standardwerte für Funktionsparameter

```
function createRandomPassword() {
   return 'n88m%an!_';
}

function createUser(name, password = createRandomPassword(), admin = false) {
   console.log('Created%s user: %s', admin ? ' admin' : '', name);
   console.log(password);
}

createUser('doejane');
createUser('bettingc', '1234');
createUser('admin', 'h o   ch s    chu  l e t   rie r', true);
```

Funktionen mit variabler Anzahl von Parametern (Rest-Parameter)

- Sprachfeature für variadische Funktionen: Rest-Parameter
  - `…args`
- Variablennamen (hier: `args`) kann gewählt werden
- Muss immer am Ende einer Parameterliste stehen
- Echtes Array-Objekt, das alle Argumente des Aufrufs entgegennimmt, für die kein Parameter vor dem Rest-Parameter existiert

```
function sum(...args) {
   let result = 0;
   for (let i = 0; i < args.length; i++) {
      result += args[i];
   }
   return result;
}
console.log(sum(1, 2, 3));              // 6
console.log(sum(1, 2, 3, 4, 5));        // 15

function printSum(prefix, ...args) {
   let result = 0;
   for (let i = 0; i < args.length; i++) {
      result += args[i];
   }
   console.log('%s%d', prefix, result);
}
printSum('Sum: ', 1, 2, 3, 4, 5);
```

Spread-Operator

- Beim Aufruf einer Methode sollen die Elemente eines Arrays als Funktionsparameter eingesetzt werden.
- Naiver Ansatz: Zugriff über Index-Operator für jedes Argument

```
function createUser(name, password, admin = false) {
   console.log('Created%s user: %s', admin ? ' admin' : '', name);
}

const user = ['admin', 'h o   ch s    chu  l e t   rie r', true];

createUser(user[0], user[1], user[2]);
```

Spread-Operator

- Sprachfeature: Spread-Operator
  - `...args`
- Die Elemente des Arrays `args` werden auf die Funktionsargumente verteilt.
  - Überflüssige Array-Elemente werden ignoriert (sind aber im `arguments`-Property enthalten)
  - Parameter ohne übergebenes Argument: `undefined`

```
function createUser(name, password, admin = false) {
   console.log('Created%s user: %s', admin ? ' admin' : '', name);
}

const user1 = ['bettingc', '1234'];
const user2 = ['admin', 'h o   ch s    chu  l e t   rie r', true];
const user3 = ['wursth', 'abcd', false, 'hans.wurst@mail.tld'];
const user4 = ['doej'];

createUser(...user1);
createUser(...user2);
createUser(...user3);
createUser(...user4);
```

Spread-Operator

- Kann auch in Array-Literalen eingesetzt werden
  - (Flaches) Kopieren
  - Konkatenieren
- Kann auch in Objekt-Literalen eingesetzt werden
  - Zusammenführung der Eigenschaften

```
const a1 = [1, 2, 3];
const a2 = [...a1];   // alt. to a1.slice()
a2.push(4);

console.log(a1, a2);
// Array(3) [1, 2, 3]
// Array(4) [1, 2, 3, 4]

const a3 = [...a1, ...a2];   // alt. to. a1.concat(a2)
console.log(a3);
// Array(7) [1, 2, 3, 1, 2, 3, 4]

const person = { name: 'C. Bettinger', gender: 'male' };
const tools = { computer: 'Mac Studio', editor: 'VS Code' };
const coder = { ...person, ...tools };
        // alternative to Object.assign({}, person, tools)

console.log(coder);
// Object {name: 'C. Bettinger', gender: 'male',
//       computer: 'Mac Studio', editor: 'VS Code'}
```

Ausführungskontext

- Innerhalb einer Funktion verweist `this` auf ein Objekt, den sog. Ausführungskontext
- `this` wird dynamisch beim Funktionsaufruf ausgesetzt
  - Aufruf als Funktion
  - Aufruf als Methode
  - Aufruf mit explizitem Kontext
  - Aufruf als Konstruktorfunktion
    - Siehe nächste Vorlesungseinheit

```
function getName() {
   return this.name;
}

const heinz = {
   name: 'Heinz',
   getName: getName
};

const anna = {
   name: 'Anna',
   getName: getName
};

console.log(getName());          // ?
name = 'Michael';                // globale Variable
console.log(getName());          // ?
console.log(heinz.getName());    // ?
console.log(anna.getName());     // ?
```

Ausführungskontext

- Häufige Fehlerquelle, da Kontext nicht dem erwarteten entspricht
  - z.B. bei Callbacks/Event Listener

- Hier:
  - `button.click()` ruft anonyme Funktion als Methode von `button` auf
  - Kontext der anonymen Funktion ist das Objekt `button`, d.h. `this` verweist innerhalb der anonymen Funktion auf das Objekt `button`
  - `button.name` ist undefiniert

```
const button = {
  onClick: null,
  click: function () {
    if (typeof this.onClick === 'function') {
      this.onClick();
    }
  }
};

function main () {
  this.name = 'main';
  button.onClick = function () {
    console.log('Click handled by: %s', this.name);
  }; // Click handled by: undefined
  button.click();
}
main();
```

Ausführungskontext

- Aufruf von `bind()` auf einer Funktion erzeugt neues Funktionsobjekt mit identischer Funktionalität, aber explizit vorgegebenem Kontext
  - Kontext wird als Argument übergeben
  - Funktion wird nicht aufgerufen

```
const button = {
  onClick: null,
  click: function () {
    if (typeof this.onClick === 'function') {
      this.onClick();
    }
  }
};

function main () {
  this.name = 'main';
  button.onClick = function () {
    console.log('Click handled by: %s', this.name);
  }.bind(this); // Click handled by: main
  button.click();
}
main();
```

Ausführungskontext

- Aufruf von `call()` auf einer Funktion führt diese Funktion unmittelbar aus
  - Kontext wird als erstes Argument übergeben
  - Parameter der Funktion können ggf. als folgende Argumente übergeben werden

- Aufruf von `apply()` auf einer Funktion führt diese Funktion unmittelbar aus
  - Kontext wird als erstes Argument übergeben
  - Parameter der Funktion können ggf. in einem Array zusammengefasst als zweites Argument übergeben werden

Arrow Functions

- Kurzschreibweise zur Definition anonymer Methoden ohne Schlüsselwort `function`
- Verwendung üblicherweise bei Übergabe von Funktionen als Argument an andere Funktionen, z.B. als Callback
- Allgemeine Syntax:
  `(Parameter) => { Rumpf }`
- Weiter verkürzte Syntax in Abhängigkeit von Anzahl der Parameter und Anweisungen

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

console.log(numbers.filter(function (x) {
   return x % 2 === 0;
}));

console.log(numbers.filter(x => x % 2 === 0));
```

Arrow Functions

```
const alwaysTrue = () => true;          // No parameter, single statement
console.log(alwaysTrue());              // true

const pow2 = x => Math.pow(x, 2);       // One parameter, single statement
console.log(pow2(4));                   // 16

const sum = (x, y) => x + y;            // Multiple parameter, single statement
console.log(sum(2, 3));                 // 5

const length = (x, y) => {              // Multiple parameter, multiple statements
   const x2 = x * x;
   const y2 = y * y;
   return Math.sqrt(x2 + y2);
};
console.log(length(3, 4));              // 5
```

Arrow Functions

- Arrow Functions erstellen keinen eigenen Funktionskontext
- Die Referenz `this` verweist innerhalb einer Arrow Function auf das gleiche Objekt wie im umschließenden Kontext
- Verzicht auf Erstellung gebundener Funktionen mithilfe von `bind()`

```
const button = {
  onClick: null,
  click: function () {
    if (typeof this.onClick === 'function‘) { this.onClick(); }
  }
};

function main () {
  this.name = 'main‘;
  // button.onClick = function () {
  //   console.log('Click handled by: %s', this.name);
  // }; // Click handled by: undefined
  button.onClick = () => {
    console.log('Click handled by: %s', this.name);
  }; // Click handled by: main
  button.click();
}
main();
```

Kontrollstrukturen

Sequenz

- Anweisungen werden durch Semikola getrennt
  - Anschließender Zeilenumbruch i.d.R. empfehlenswert
- *Automatic Semicolon Insertion*
  - Führt gelegentlich zu überraschenden Effekten
  - Laut Spezifikation eine Maßnahme zur Fehlerkorrektur
    - Erhöhung der Toleranz ggü. Programmierfehlern(!)
  - https://brendaneich.com/2012/04/the-infernal-semicolon/

```
console.log('Hello, World!')
console.log('Hello, World!');
console.log('Hello, '); console.log('World!')
```

Bedingte Verzweigung: if

```
const h = new Date().getHours();
console.log('Hour: %d', h);

if (h < 9) {
    console.log('Morning');
}
else if (h < 12) {
    console.log('Forenoon');
}
else {
    console.log('Time to wake up');
}
```

Bedingte Verzweigung: if

- Erinnerung: Alle Werte können zu Wahrheitswerten evaluiert werden
  - Ermöglicht oft kompaktere Schreibweise

```
function log(msg) {
   if (msg !== undefined && msg !== null && msg.length !== 0) {
      console.log(msg);
   }
}

function log2(msg) {
   if (msg) { console.log(msg); }
}

log2(undefined); log2(null); log2('');        //
log2('Hello');                                // Hello
```

Bedingte Verzweigung: switch-case

```
const month = new Date().getMonth() + 1;

switch (month)  {
   case 3:
   case 4:
   case 5:
      console.log('Spring');
      break;
   case 6:
   case 7:
   case 8:
      console.log('Summer');
      break;
   case 9:
   case 10:
   case 11:
      console.log('Autumn');
      break;
   case 12:
   case 1:
   case 2:
      console.log('Winter');
      break;
   default:
      console.error('WTF?!');
}
```

Schleifen: while, do-while

```
let i = 1;

while (i <= 50) {
   console.log(i);
   i++;
}
```

```
1
2
3
...
50
```

```
let i = 1;

do {
   console.log(i);
   i++;
} while (i <= 50);
```

```
1
2
3
...
50
```

Schleifen: for

```
for (let i = 1; i <= 50; i++) {
   console.log(i);
}
```

```
1
2
3
...
50
```

```
for (let i = 1; i <= 10; i++) {
   let line = '';
   for (let j = 1; j <= 10; j++) {
      line += ((j*i) + '\t');
   }
   console.log(line);
}
```

Ausgabe?

Schleifen: for

- Enthält das Abbruchkriterium den Zugriff auf eine Objekteigenschaft sollte dessen Wert in der Schleifeninitialisierung zwischengespeichert werden
- Erhöht Performanz, z.B. beim Iterieren über Objekte vom Typ
  - `Array`
  - `HTMLCollection`

```
const a = ['a', 'b', 'c', 'd', 'e'];

for (let m = 0, length = a.length; m < length; m++) {
   console.log(a[m]);
}
```

Schleifen: for..in

- `for..in`-Schleife iteriert über alle aufzählbaren **Eigenschaftsschlüssel eines beliebigen Objekts**
  - Iteriert in willkürlicher Reihenfolge, i.d.R. also nicht für Arrays geeignet
  - Iteriert auch über „geerbte“ Eigenschaften (genauer: Eigenschaften der Prototypen)
    - Siehe nächste Vorlesungseinheit

```
const a = ['a', 'b', 'c', 'd', 'e’];
for (const key in a) { console.log(key); }   // 0, 1, 2, 3, 4

const phonebookEntry = { firstName: 'Hans‘, lastName: 'Wurst‘, number: '0123-456789‘, print: function() {…} };
for (const key in phonebookEntry) {
   console.log('%s: %s (%s)', key, phonebookEntry[key], typeof phonebookEntry[key]);
}
```

Schleifen: for..of

- `for..of`-Schleife iteriert über alle aufzählbaren **Eigenschaftswerte eines iterierbaren Objekts**
  - Iterierbare Objekte sind Objekte, die eine bestimmte Methode implementieren, die wiederrum einen sog. Iterator zurückliefert, z.B.
    - `Array`
    - `Map`
    - `Set`
    - Aber insbesondere nicht: `Object`

```
const a = ['a', 'b', 'c', 'd', 'e'];
for (const value of a) { console.log(value); }   // a, b, c, d, e
```

Schleifen: continue

- Schlüsselwort `continue` springt unmittelbar zur nächsten Iteration

```
for (let i = 1900; i <= 2000; i++) {
   if (((i % 4 === 0) && (i % 100 !== 0)) || (i % 400 === 0)) {
      continue;
   }
   console.log(i);
}
```

Ausgabe?

Schleifen: break

- Schlüsselwort `break` springt unmittelbar hinter die aktuelle Schleife

```
function search(haystack, needle) {
   let found = false;
   for (let i = 0, length = haystack.length; i < length; i++) {
      if (haystack[i] === needle) {
         found = true;
         break;
      }
   }
   return found;
}

console.log(search([1, 2, 3, 4, 5], 3));
```

Fehlerbehandlung

Fehler werfen: throw

- Laufzeitfehler können mithilfe des Schlüsselworts `throw` geworfen werden

```
throw new Error('Error message');
```

- Objekt hinter `throw` sollte von einem Standardfehlertyp oder von einem eigens davon abgeleiteten Typ sein
  - Grundsätzlich können sogar beliebige Objekte geworfen werden

- Standardfehlertypen
  - `Error`
    - `EvalError`
    - `SyntaxError`
    - `RangeError`
    - `TypeError`
    - `ReferenceError`
    - `URIError`

Fehler abfangen: try-catch-finally

- Fehlerabhang mithilfe eines `try-catch-finally`-Konstrukts
- Nur ein `catch`-Block
- `finally`-Block ist optional und wird immer ausgeführt, unabhängig davon, ob innerhalb des `try`-Blocks ein Fehler geworfen wurde oder nicht

```
try {
  // do something that may
  // throw an error, e.g.
  const i = 42;
  console.log(i.toUpperCase());
}
catch (error) {
  console.error('Unable to print i:', error.message);
}
finally {
  console.log('Done.');
}
```

Weitere Datenstrukturen

Map

- `Map`: Datenstruktur zur Sammlung von Schlüssel-Wert-Paaren
  - Vergleichbar mit assoziativen Arrays (`Object`)
  - Methoden, Unterschiede und Nutzungsempfehlungen (https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Map)
    - Wenn als Schlüssel nicht nur Strings sondern beliebige Objekte in Frage kommen
    - Wenn Reihenfolge der Schlüssel-Wert-Paare beibehalten werden soll
    - Wenn die Größe der Sammlung abgefragt werden muss
    - Wenn Schlüssel-Wert-Paare oft eingefügt und gelöscht werden müssen
      - Methoden von `Map` sind i.d.R. laufzeiteffizienter

Map

```
const salaries = new Map([
   ['Jane Doe', 55000],
   ['Hans Wurst', 32000]
]);

console.log(salaries.size);                          // 2
console.log(salaries.get('Hans Wurst'));             // 32000
salaries.set('Max Mustermann', 44000);
console.log(salaries.size);                          // 3
console.log(salaries.has('Max Mustermann'));         // true
salaries.delete('Jane Doe');
console.log(salaries.size);                          // 2
salaries.clear();
console.log(salaries.size);                          // 0
```

Set

- Datenstruktur zur Repräsentation einer Menge von Schlüssel-Werten
  - Ein Wert ist höchstens einmal in der Menge enthalten
  - Gleichheitsprüfung via ===-Operator

```
const employees = new Set([
   'Jane Doe',
   'Hans Wurst',
   'Jane Doe'
]);

console.log(employees.size);  // 2 (!)

console.log(employees.has('Max Mustermann'));  // false

employees.add('Max Mustermann');
employees.delete('Jane Doe');
console.log(employees.size);  // 2

employees.clear();
console.log(employees.size);  // 0
```

WeakMap und WeakSet

- `WeakMap`- bzw. `WeakSet-Objekte` referenzieren die Schlüssel-Objekte nur schwach
- Gibt es außerhalb der `WeakMap` bzw. des `WeakSet` keine weitere Referenz auf diese Objekte, können diese vom Garbage Collector gelöscht werden
- Weitere Unterschiede:
  - Als Objekte kommen bei `WeakMap` keine primitiven Datentypen in Frage
  - Die Methoden `keys()`, `values()`, `entries()`, `clear()` sowie die Eigenschaft `size` fehlen

Iteratoren

- Die Methoden `keys()`, `values()` und `entries()` von `Map`, `Set` und `Array` (aber nicht `Object`) liefern sog. Iteratoren zurück
  - Vereinfachen das (reihenfolgetreue) Iterieren über Datenstrukturen mithilfe der `for..of`-Schleife
- `keys()`: Iterator für die Schlüssel
- `values()`: Iterator für die Werte (`Set`: identisch mit `keys()`)
- `entries()`: Iterator über Schlüssel-Wert- bzw. Wert-Wert-Paare

Iteratoren

```
const salaries = new Map([
   ['Jane Doe', 55000],
   ['Hans Wurst', 32000],
   ['Max Mustermann', 44000]
]);

for (const name of salaries.keys()) {
   console.log(name);                                    // Jane Doe, Hans Wurst, …
}

for (const salary of salaries.values()) {
   console.log(salary);                                  // 55000, 32000, …
}

for (const employee of salaries.entries()) {
   console.log('%s: %d', employee[0], employee[1]);      // Jane Doe: 55000, ...
}
```

Iteratoren

```
const names = ['Jane Doe', 'Hans Wurst', 'Max Mustermann'];

for (const index of names.keys()) {
   console.log(index);                              // 0, 1, …
}

for (const name of names.values()) {
   console.log(name);                               // Jane Doe, Hans Wurst, …
}

for (const name of names.entries()) {
   console.log('%s: %s', name[0], name[1]);         // 0: Jane Doe, …
}
```

Iterierbare Objekte

- Die Objekte `Map`, `Set`, `Array` (aber nicht `Object`) sind sog. iterierbare Objekte
- Erlauben `for..of`-Schleife auf den Objekten selbst
  - `Map: entries()`
  - `Set: values()`
  - `Array: values()`

```
const salaries = new Map([…]);
const names = ['Jane Doe', 'Hans Wurst', 'Max Mustermann'];
const employees = new Set(names);

for (const employee of salaries) {
   console.log(employee);
}

for (const employee of employees) {
   console.log(employee);
}

for (const name of names) {
   console.log(name);
}
```

Object

- Methoden des Standardobjekts `Object`
  - `Object.keys()`
  - `Object.values()`
  - `Object.entries()`

```
const employee = {
   firstName: 'Hans',
   lastName: 'Wurst',
   salary: 33000,
   car: {
      model: 'Opel Kadett',
      buildYear: 1987
   }
};

console.log(Object.keys(employee));
// ['firstName', 'lastName', 'salary', 'car']

console.log(Object.values(employee));
// ['Hans', 'Wurst', 33000, Object]

console.log(Object.entries(employee));
// [Array(2), Array(2), Array(2), Array(2)]
```

Object

- Methoden des Standardobjekts `Object`
  - `Object.fromEntries()`
  - `Object.assign()`

```
const entries = [['a', 'b'], ['c', 'd']];
const map = new Map(entries);

console.log(Object.fromEntries(entries));
// Object { a: 'b', c: 'd' }
console.log(Object.fromEntries(map));
// Object { a: 'b', c: 'd' }

const person = { name: 'C. Bettinger', gender: 'male' };
const tools = { computer: 'Mac Studio', editor: 'VS Code' };

//const coder = { ...person, ...tools };
const coder = Object.assign({}, person, tools);
console.log(coder);
// Object {name: 'C. Bettinger', gender: 'male',
//       computer: 'Mac Studio', editor: 'VS Code'}
```

Object

- Methoden des Standardobjekts `Object`
  - `Object.seal()`
    - Verhindert Hinzufügen oder Löschen von Eigenschaften
  - `Object.freeze()`
    - Verhindert zusätzlich die Änderung von Eigenschaftswerten

```
const employee = { ... };

Object.seal(employee);
employee.gender = 'male';
employee.salary = 0;
delete employee.car;
console.log(employee);
// Object {firstName: 'Hans', lastName: 'Wurst',
//       salary: 0, car: Object}

Object.freeze(employee);
employee.salary = 10000;
console.log(employee.salary);
// 0

const Color = Object.freeze({
   RED: 0xFF0000,
   GREEN: 0x00FF00,
   BLUE: 0x0000FF
});
```

Object

- Methoden des Standardobjekts `Object`
  - `Object.hasOwn()`
    - Prüft, ob ein Objekt eine eigene Eigenschaft mit bestimmtem Namen hat
    - Im Gegensatz zum `in`-Operator wird dabei die sog. Prototypenkette nicht beachtet

```
console.log(Object.hasOwn(employee, 'lastName‘));
         // true
console.log(Object.hasOwn(employee, 'familyName‘));
         // false

console.log(Object.hasOwn(employee, 'toString‘));
         // false
console.log('toString' in employee);
         // true
```

Destrukturierung von Arrays

- Destrukturierung: Zuweisung einzelner Array- oder Objekteigenschaften an Variablen
- Hier: Die Elemente des Arrays `sortedNames` werden an die Variablen `first`, `second`, `third`, `fourth`, `fifth` und `sixth` zugewiesen
- Array-Elemente können durch Weglassen einer Variablen ignoriert werden
- Gibt es kein entsprechendes Array-Element bekommt die Variable den Wert `undefined` (hier: `fifth`).
- Es können Standard-Werte angeben werden (hier für `sixth`)
- Funktioniert auch mit mehrdimensionalen Arrays

```
const sortedNames = ['Hans Wurst', 'Max Mustermann‘,
    'Jane Doe', 'John Doe'].sort();

// const first = sortedNames[0];
// const second = sortedNames[1];
// const third = sortedNames[2];

const [first, second, third] = sortedNames;
console.log(first, second, third);

const [, , , fourth, fifth] = sortedNames;
console.log(fourth, fifth);

const [, , , , , sixth = 'Ivan Ivanovitch‘]
    = sortedNames;
console.log(sixth);
```

Destrukturierung von Arrays

```
const salaries = new Map([
  ['Jane Doe', 55000],
  ['Hans Wurst', 32000],
  ['Max Mustermann', 44000]
]);

//for (const salary of salaries) {
//    console.log(`${salary[0]}: ${salary[1]} USD`);
//}

for (const [name, salary] of salaries) {
    console.log(`${name}: ${salary} USD`);
}
```

Destrukturierung von Objekten

```
const employee = {
   firstName: 'Hans',
   lastName: 'Wurst',
   salary: 33000,
   car: {
      model: 'Opel Kadett',
      buildYear: 1987
   }
};

const {
   firstName, // abbrev. for 'firstName: firstName'
   lastName,
   car: {
      buildYear
   }
} = employee;

console.log('%s %s: %d', lastName, firstName, buildYear);
```

```
const employees = [ {
   firstName: 'Hans',
   lastName: 'Wurst',
   salary: 33000,
   car: {
      model: 'Opel Kadett',
      buildYear: 1987
   }
}, … ];

const currentYear = new Date().getFullYear();

for (const { firstName, lastName, car: { buildYear } } of employees) {
   console.log('%s %s, age of car: %d', lastName, firstName,
         currentYear - buildYear);
}
```

Promises und async/await

Asynchrone Funktionen via Callback

- Wesentliches Konzept der Web-Entwicklung: Asynchrone Programmierung
- Asynchrone Funktion: Ausführung einer Funktion kann länger dauern, z.B.
  - Einlesen von Dateien auf der Festplatte
  - Nachladen dynamischer Webseiten-Inhalte
- Aufrufer der Funktion wartet aber nicht mit der Ausführung der nächsten Anweisung auf das Ende der asynchronen Funktion
- Stattdessen: Entwurfsmuster *Callback*
  - Übergeben einer Funktion an die asynchrone Funktion, die von dieser aufgerufen wird, sobald die asynchrone Funktion erfolgreich oder nicht erfolgreich beendet wurde

Asynchrone Funktionen via Callback

```
function asyncA(callback) {
   // do something long lasting, e.g. read
   // a file, do a HTTP request or just
   // wait for 100 milliseconds
   const value = Math.random();
   if (value < 0.5) {
      // something wrong happens
      callback(new Error(value), null);
   }
   else {
      // everything is fine, we're done
      callback(null, value);
   }
}
```

```
asyncA((error, result) => {
   if (error) {
      console.error(error);
   }
   else {
      console.log(result);
   }
});
```

Pyramid of Doom

```
function asyncA(callback) { … }   // as before

function asyncB(value, callback) {
   value *= Math.random();
   if (value < 0.25) {
      callback(new Error(value), null);
   }
   else {
      callback(null, value);
   }
}

function asyncC(value, callback) { … }   // as asyncB
function asyncD(value, callback) { … }   // as asyncB
```

```
asyncA((error, result) => {
   if (error) {
      console.error(error);
   }
   else {   // call another async function and pass result
      asyncB(result, (error, result) => {
         if (error) {
            console.error(error);
         }
         else {   // call another async function and pass result
            asyncC(result, (error, result) => {
               if (error) {
                  console.error(error);
               }
               else { // call another async function and pass result
                  asyncD(result, (error, result) => {
                     if (error) {
                        console.error(error);
                     }
                     else {
                        console.log(result);
                     }
                  });
               }
            });
         }
      });
   }
});
```

Promises

- Standardobjekt `Promise`
- Asynchrone Funktion liefert `Promise-Objekt` zurück
  - "Versprechen", welches später erfüllt oder abgelehnt werden kann
- `Promise`-Konstruktorfunktion erwartet eine Funktion, in der die asynchrone Aktivität implementiert ist
  - Aufruf der Funktionsparameter `resolve` und `reject` innerhalb der Aktivität führt zum Erfüllen bzw. Ablehnen des Versprechens
    - An `resolve` sollte ggf. Ergebnis/Rückgabe übergeben werden
    - An `reject` sollte `Error`-Objekt übergeben werden
- `Promise-Methoden then()` und `catch()` erwarten Funktionen, in denen auf die Erfüllung bzw. die Ablehnung des Versprechens reagiert werden kann

Asynchrone Funktionen via Promise

```
function asyncA() {
   return new Promise((resolve, reject) => {
      const value = Math.random();
      if (value < 0.5) {
         reject(new Error(value));
      }
      else {
         resolve(value);
      }
   });
}
```

```
asyncA().then(value => {
   console.log(value);
}).catch(reason => {
   console.error(reason);
});
```

Verkettung von Promises

- Die Methoden `then()` und `catch()` geben immer selbst ein `Promise-Objekt` zurück
  - Explizit (im Beispiel)
  - Implizit (falls die an `then()` übergebene Funktion einen Ausdruck zurück gibt)
- Erlaubt Verkettung von Promises sowie deren Behandlung

```
function asyncA() { … }   // as before

function asyncB(value) {
   return new Promise((resolve, reject) => {
      value *= Math.random();
      if (value < 0.05) {
         reject(new Error(value));
      } else {
         resolve(value);
      }
   });
}
function asyncC(value) { … }   // as asyncB
function asyncD(value) { … }   // as asyncB

asyncA().then(asyncB).then(asyncC).then(asyncD).then(value => {
   console.log(value);
}).catch(reason => {
   console.error(reason)
});
```

Promises: finally()

- Häufig gleiche abschließende Aktionen unabhängig von Erfolgs- oder Fehlerfall notwendig
- `Promise`-Methode `finally()` erwartet parameterlose Funktion und wird nach letztem `then()`- bzw. `catch()`-Handler aufgerufen

```
const doSomething = new Promise(...);

const button = document.getElementById('okButton');
button.addEventListener('click', () => {
   button.disabled = true;

   doSomething.then(() => {
      console.log('yay');
   }).catch(() => {
      console.error('oops');
   }).finally(() => {
      button.disabled = false;
   });
});
```

Promise.race() und Promise.all()

- Statische Methoden
  - Erwarten Array von `Promise`-Objekten
  - Geben ein `Promise`-Objekt zurück
- `Promise.race()`
  - Erfüllt, sobald ein übergebenes `Promise`-Objekt erfüllt wurde
  - Abgelehnt, falls kein übergebenes `Promise`-Objekt erfüllt wurde
- `Promise.all()`
  - Erfüllt, falls alle übergebenen `Promise`-Objekte erfüllt wurden
  - Abgelehnt, sobald ein übergebenes `Promise`-Objekt abgelehnt wurde

```
function asyncA(callback) { … }
function asyncB(callback) { … }
function asyncC(callback) { … }
function asyncD(callback) { … }

Promise.race([asyncA(), asyncB(), asyncC(), asyncD()]).then(value => {
   console.log(value);
}).catch(reason => {
   console.error(reason);
});

Promise.all([asyncA(), asyncB(), asyncC(), asyncD()]).then(values => {
   console.log(values);
}).catch(reason => {
   console.error(reason);
});
```

async/await

- Schlüsselwörter `async` und `await` in ES8
- Syntaktische Ergänzung von Promises zur weiteren Vereinfachung von asynchronen Programmabläufen

async/await

- Schlüsselwort `await` wartet auf ein nachgestelltes `Promise`-Objekt und
  - gibt den Wert zurück, falls `Promise` erfüllt wird
  - wirft einen Fehler mit dem entsprechenden Wert, falls `Promise` abgelegt wird
    - Behandlung mit `try-catch`-Block
- Darf nur innerhalb einer Funktion, die mit dem Schlüsselwort async gekennzeichnet ist, verwendet werden

```
function asyncA() {
  return new Promise((resolve, reject) => {
    const value = Math.random();
    if (value < 0.5) {
      reject(new Error(value));
    }
    else {
      resolve(value);
    }
  });
}

(async function () {
  try {
    const value = await asyncA();
    console.log(value);
  }
  catch (reason) {
    console.error(reason);
  }
})();
```

async/await

```
function asyncA() { … }          // as before
function asyncB(value) { … }     // as before
function asyncC(value) { … }     // as asyncB
function asyncD(value) { … }     // as asyncB

// asyncA().then(asyncB).then(asyncC).then(asyncD).then(value => {
//    console.log(value);
// }).catch(reason => {
//    console.error(reason)
// });
```

```
(async function () {
   try {
      let value = await asyncA();
      value = await asyncB(value);
      value = await asyncB(value);
      value = await asyncD(value);
      console.log(value);
   }
   catch (reason) {
      console.error(reason);
   }
})();
```

async/await

```
// anonymous function
document.body.addEventListener('click', async function () { const value = await fetch('/‘); … });
document.body.addEventListener('click', async () => { const value = await fetch('/‘); … });

// object property
const obj = { async method() { const value = await fetch('/'); } };

// class methods
class MyClass { async method() { const value = await fetch('/'); } }

// awaiting multiple promises
await Promise.all([downloadA(), downloadB()]);
```

Fragen?