Web-Entwicklung

ECMAScript 2015 und
neuere Versionen (ES6+)

Inhalte der Vorlesung

- ECMAScript 2015 und neuere Versionen (ES6+): Ausgewählte Neuerungen
  - Variablen und Konstanten
  - Funktionen
  - Klassensyntax
  - Datenstrukturen
  - Promises und async/await
  - Weitere Neuerungen

Browserkompatibilität

- Je neuer das Feature, desto größer ist die Wahrscheinlichkeit, dass nicht alle (wichtigen) Browser es unterstützen
- Einzelfallprüfung unter Abwägung der Zielplattformen wichtig, z.B. auf https://caniuse.com
- Hier: Angabe, falls keine Unterstützung durch aktuelle Desktop-Browser
  - Chrome, Firefox, Safari, Opera, Edge
  - Nicht: IE

Variablen und Konstanten

Variablen mit block scope (let)

- Die Gültigkeit einer Variablen, die mit `var` deklariert wurde, wird nicht durch Blöcke begrenzt (*block scope*), sondern durch Funktionen (*function level scope*)
  - Innerhalb einer Funktion definierte Variablen sind in der gesamten Funktion gültig
- Neues Schlüsselwort `let` erlaubt die Deklaration von Variablen, die nur in dem Block, in dem sie deklariert wurden, gültig sind
  - z.B. nur innerhalb eines Schleifenrumpfes
  - Erlaubt auch mehrfache Deklaration der gleichen Variable innerhalb einer Funktion (Überdeckung)
    - Sollte vermieden werden!
  - Verwendung wie `var`
- Empfehlung: Weitestgehender Verzicht auf `var` zugunsten von `let`

Variablen mit block scope (let)

```
let i = 4711;

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

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

Konstanten (const)

- Neues Schlüsselwort `const` erlaubt die Deklaration von Konstanten
  - Deklaration und Initialisierung muss in einer Anweisung erfolgen
  - Anschließende Zuweisungen erzeugen Laufzeitfehler
    - Verhindert nicht die Änderung von Objekteigenschaften
  - Konvention: Nur Großbuchstaben, Worttrennung mit Unterstrich

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

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

Template Strings

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

```
let name = "Heinz";
let age = 42;

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

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

console.log(text);
```

Funktionen

Standardwerte für Funktionsparameter

- Neues Sprachfeature erlaubt Angabe von Standardwerten für Funktionsparameter
  - Ermöglicht optionale Argumente ohne Überladung und ohne manuelle Überprüfung im Funktionsrumpf
- 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("bettingc", "1234");
createUser("admin", "h o   ch s    chu  l e t   rie r", true);
createUser("doejane");
```

Optionale Funktionsargumente via Standardwert

```
function log(message, data = null) {
   console.log(message);

   if (data) {
      console.dir(data);
   }
}

log("Simple log message");
log("Log message with payload", { error: false, line: 42 });
```

Funktionen mit variabler Anzahl von Parametern (Rest-Parameter)

- Variadische Funktionen: Funktionen, die beliebig viele Argumente erwarten
  - Beispiel: Funktion zur Berechnung der Summe beliebig vieler Argumente
- Problematisch
  - `arguments` ist kein echtes Array-Objekt, besitzt also nicht alle Funktionen
  - Funktionssignatur gibt keinen Hinweis darauf, ob die Funktion kein Argument oder beliebig viele Argumente erwartet

```
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
```

Funktionen mit variabler Anzahl von Parametern (Rest-Parameter)

- Neues Sprachfeature: 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);
}

let user = ["admin", "h o   ch s    chu  l e t   rie r", true];

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

Spread-Operator

- Neues 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);
}

let user1 = ["bettingc", "1234"];
let user2 = ["admin", "h o   ch s    chu  l e t   rie r", true];
let user3 = ["wursth", "abcd", false, "hans.wurst@mail.tld"];
let user4 = ["doej"];

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

Spread-Operator

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

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

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

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

let person = { name: "C. Bettinger", gender: "male" };
let tools = { computer: "iMac Pro", editor: "VS Code" };
let coder = { ...person, ...tools };

console.log(coder);
// Object {name: "C. Bettinger", gender: "male",
//       computer: "iMac Pro", editor: "VS Code"}
```

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

```
let 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

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

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

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

let length = (a, b) => {                  // Multiple parameter, multiple statements
   let a2 = a * a;
   let b2 = b * b;
   return Math.sqrt(a2 + b2);
};
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()`

```
function Button(onClick = null) {
   this.onClick = onClick;
}
Button.prototype.click = function () {
   if (typeof this.onClick === "function") {
      this.onClick();
   }
};
(function () {
   this.name = "main";

   //let button = new Button(function () {
   //   console.log("Click handled by: %s", this.name);
   //});    // Click handled by: undefined

   let button = new Button(() => console.log("Click
         handled by: %s", this.name));
                        // Click handled by: main

   button.click();
})();
```

Klassensyntax

Klassensyntax

- Populärste Neuerung: Alternative Syntax zur Realisierung pseudoklassischer Vererbung
  - Schlüsselwörter und Syntax lehnen sich an gewohnte Konstrukte aus Java/C++/C# an
    - Klassen mit Konstruktoren, Statische Methoden
    - `class, extends, constructor, super, static`
  - Zur Laufzeit sind Datenstrukturen aber identisch mit den in der letzten Vorlesungseinheit gezeigten
    - Es gibt weiterhin nur Objekte – keine Klassen!

Klasse, Konstruktor, Methode

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   print() {
      console.log("%s %s", this.firstName, this.lastName);
   }
}

let p1 = new Person("Hans", "Wurst");
p1.print();

let p2 = new Person("Max", "Mustermann");
p2.print();

new Person("Eva", "Meier").print();
```

Vererbung und Überschreiben von Methoden

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   print() {
      console.log("%s %s", this.firstName,
            this.lastName);
   }
}

class Employee extends Person {
   constructor(firstName, lastName, salary = 48000) {
      // call superclass constructor
      super(firstName, lastName);
      this.salary = salary;             // new property
   }

   print() {               // override method
      super.print();       // calling overriden method
      console.log("Salary: %s", this.salary);
   }
}
```

```
let p1 = new Person("John", "Smith");
p1.print();

let e1 = new Employee("Jane", "Doe", 55000);
e1.print();

class Animal {}

console.log(e1 instanceof Employee);  // true
console.log(e1 instanceof Person);        // true
console.log(e1 instanceof Animal);        // false
```

Statische Eigenschaften

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;

      Person.COUNT++;
   }

   print() { console.log("%s %s", this.firstName, this.lastName); }

   static printCount() { console.log("Number of Persons: %d", Person.COUNT); }      // static method
}

Person.COUNT = 0;                  // static property as property of "class" (=constructor function)

class Employee extends Person { ... }

let p1 = new Person("John", "Smith");
let e1 = new Employee("Jane", "Doe", 55000);

Person.printCount();               // ?
```

Quasi-Private Eigenschaften

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   getFirstName() { return this.firstName; }

   setFirstName(value) {
      if (value.length !== 0) {
         this.firstName = value;
      }
   }

   getLastName() { return this.lastName; }

   setLastName(value) {
      if (value.length !== 0) {
         this.lastName = value;
      }
   }
   print() {
      this._log();
   }

   _log() {
      console.log("%s %s", this.firstName,
            this.lastName);
   }
}



let p = new Person("Hans", "Wurst");
p.setFirstName("John");
p.print();                              // John Wurst

console.log(p instanceof Person);       // true
```

Getter und Setter

- Neues Sprachkonstrukt zur Definition von Getter-/Setter-Methoden
  - Einführung bereits mit ES5, Ausbau in ES6
- Schlüsselwörter `get` bzw. `set` vor Funktionsdefinitionen
- Lesender und schreibender Zugriff auf Eigenschaften wird in Funktionen gekapselt
  - Alternative zu normalen Methoden mit Bezeichnerpräfix "`get`" bzw. "`set`"
- Aber: Verwendung wie eine Dateneigenschaft, nicht wie eine Methode
  - Zugrundeliegende Eigenschaft muss anderen Namen tragen – sonst Endlosrekursion!

Getter und Setter

```
class Person {
     constructor(firstName, lastName) {
        this._firstName = firstName;
        this._lastName = lastName;
     }

     get firstName() { return this._firstName; }

     set firstName(value) {
        if (value.length !== 0) { this._firstName = value; }
     }

     get lastName() { return this._lastName; }

     set lastName(value) {
        if (value.length !== 0) { this._lastName = value; }
     }

     print() { … }
}
```

```
let p = new Person("Hans", "Wurst");
p.firstName = "John";
p.print();                      // John Wurst
console.log(p.lastName);        // Wurst
```

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 (`Map.prototype.size`)
    - 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

```
let 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 (Gleicher Wert und gleicher Datentyp)

```
let 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
    - i.d.R. mithilfe der ebenfalls neuen `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

Schleifen: for..of

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

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

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

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

Schleifen: for..of

```
let names = ["Jane Doe", "Hans Wurst", "Max Mustermann"];

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

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

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

Schleifen: for..of

- 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()`

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

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

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

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

Iteratoren und iterierbare Objekte

- Auch selbst entwickelte Objekte können iterierbar sein. Dazu müssen folgende Konventionen eingehalten werden:
- Iterierbares Objekt: Methode `[Symbol.iterator]` liefert Iterator zurück
- Iterator: Objekt mit Methode `next()`, die sich wie folgt verhält:
  - Wurden noch nicht alle Elemente in der Datenstruktur besucht, liefert die Methode ein Objekt mit den Schlüsseln `value` und `done` aus. Der Wert zum Schlüssel `value` ist das nächste Element in der Datenstruktur. `done` enthält den Wert `false`.
  - Wurden alle Elemente in der Datenstruktur besucht, liefert die Methode ein Objekt mit dem Schlüssel `done` und dem Wert `true`.

Iteratoren und iterierbare Objekte

```
class ReverseSortedArray {
   constructor(array) {
      this.array = array.sort();
   }

   [Symbol.iterator]() {
      let currentIndex = this.array.length - 1;
      return {
         next: () => {
            if (currentIndex >= 0) { return {
               value: this.array[currentIndex--],
               done: false };
            }
            else { return { done: true }; }
         }
      };
   }
}

module.exports = ReverseSortedArray;
```

Iteratoren und iterierbare Objekte

```
let ReverseSortedArray = require("./ReverseSortedArray");

let names = new ReverseSortedArray(["Ross Melanie", "Hill Patsy", "Hicks Peggy", "Higgins Dixie", "Cortez Nettie", "Austin
      Gertrude", "Garza Fred", "Beck Katie", "Cummings Larry", "Reid Erin"]);

for (let name of names) {
   console.log(name);           // Ross Melanie
}                               // Reid Erin
                                // Hill Patsy
                                // …
                                // Beck Katie
                                // Austin Gertrude
```

Object

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

```
let employee = {
   firstName: "Hans",
   lastName: "Wurst",
   salary: 33000,
   car: {
      model: "Ford Taurus",
      buildYear: 1999
   }
};

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

- Neue Methoden des Standardobjekts `Object`
  - `Object.fromEntries()`
    - Keine Unterstützung in Edge
  - `Object.assign()`

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

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

let person = { name: "C. Bettinger", gender: "male" };
let tools = { computer: "iMac Pro", editor: "VS Code" };

//let coder = { ...person, ...tools };
let coder = Object.assign({}, person, tools);
console.log(coder);
// Object {name: "C. Bettinger", gender: "male",
//       computer: "iMac Pro", editor: "VS Code"}
```

Object

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

```
let 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
});
```

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

```
let sortedNames = new
   ReverseSortedArray(["Ross Melanie", "Hill Patsy",
      "Hicks Peggy", "Higgins Dixie"]).toArray();


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

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

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

let [, , , , , sixth = "Jane Doe"] = sortedNames;
console.log(sixth);
```

Destrukturierung von Arrays

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

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

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

Destrukturierung von Objekten

```
let employee = {
   firstName: "Hans",
   lastName: "Wurst",
   salary: 33000,
   car: {
      model: "Ford Taurus",
      buildYear: 1999
   }
};

let {
   firstName, // abbrev. for "firstName: firstName"
   lastName,
   car: {
      buildYear
   }
} = employee;

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

```
let employees = [ {
   firstName: "Hans",
   lastName: "Wurst",
   salary: 33000,
   car: {
      model: "Ford Taurus",
      buildYear: 1999
   }
}, … ];

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

for (let { firstName, lastName, car: { buildYear } } of employees) {
   console.log("%s %s: %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
   let value = Math.random();
   if (value < 0.1) {
      // something wrong happens
      callback(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 < 0.05) {
      callback(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

- Neues Standardobjekt `Promise`
- Asynchrone Funktion liefert `Promise-Objekt` zurück
  - "Versprechen", welches später erfüllt oder abgelehnt werden kann
- `Promise-Konstruktor` 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
- `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) => {
      let value = Math.random();
      if (value < 0.1) {
         reject(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(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)
});
```

Promise.prototype.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

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

let 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

- Neue Schlüsselwörter `async` und `await` in ES8
  - Unterstützt in allen gängigen Browsern (abgesehen von IE)
- 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) => {
    let value = Math.random();
    if (value < 0.1) {
      reject(value);
    }
    else {
      resolve(value);
    }
  });
}

(async function () {
  try {
    let 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 () {
   let value = await fetch('/');
});

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

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

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

Weitere Neuerungen

Weitere Neuerungen

- Native Module
- Neue Standardobjekte `Proxy`, `TypedArray`, …
- Neue Methoden der Standardobjekte `String`, `Array`, `RegExp`, `Number` und `Math`
- Primitiver Datentyp `Symbol`
- Erweiterungen der Objektliteral-Notation
- Benannte Funktionsparameter
- Generatorfunktionen und `yield-Statement`
- …

Fragen?