

Periode 1 Learning-Goals

<https://docs.google.com/document/d/1wkCSwyz-SFn6C0KtqVtUt9NiYGnTa81yUEHdaES8UxY/edit>

Explain and Reflect:

Explain the differences between Java and JavaScript + node.
Topics you could include:

- that Java is a compiled language and JavaScript a scripted language
- Java is both a language and a platform
- General differences in language features.
- Blocking vs. non-blocking

Generelt:

Java: Er et object-orienteret programmerings sprog. Med Java laver man compileret programmer som kan køre på næsten alle platforme.

JavaScript: et et mere lightweight programming sprog det benytter et interpreted scripting language. Det bliver brugt til at lave en del web-udvikling hvor Java typisk bliver brugt til Backend, bliver JavaScript brugt til frontend.

Type:

Java: er meget type stærkt og skal benytter static type checking. Dvs man skal angive hvilken type variable man giver videre feks int, string osv.

JavaScript: er mindre type stærkt altså dynamic typing man behøver ikke fortælle hvilken type variable man giver med videre eller vil have retur i mange tilfælde. (her kommer TypeScript in handy)

Blocking - Non-blocking

Node applikationer fungerer ved begrebet "Non-blocking - Asynchronous."

En single thread kan håndterer flere request

Java	JavaScript
Java is strongly typed language and variable must be declare first to use in program. In Java the type of a variable is checked at compile-time.	JavaScript is weakly typed language and have more relaxed syntax and rules.
Java is an object oriented programming language.	JavaScript is an object based scripting language.
Java applications can run in any virtual machine(JVM) or browser.	JavaScript code used to run only in browser, but now it can run on server via Node.js.
Objects of Java are class based even we can't make any program in java without creating a class.	JavaScript Objects are prototype based.
Java program has file extension ".Java" and translates source code into bytecodes which is executed by JVM(Java Virtual Machine).	JavaScript file has file extension ".js" and it is interpreted but not compiled, every browser has the Javascript interpreter to execute JS code.
Java is a Standalone language.	contained within a web page and integrates with its HTML content.
Java program uses more memory.	JavaScript requires less memory therefore it is used in web pages.
Java has a thread based approach to concurrency.	Javascript has event based approach to concurrency.
Java supports multithreading.	Javascript doesn't support multi-threading.

Explain generally about node.js, when it "makes sense" and *npm*, and how it "fits" into the node ecosystem.

Node:

Med JavaScript kan man benytte sig af node, Node er et cross-platform runtime system. Benyttes til udvikling af server-side webapplikationer. Node.js er skrevet i JavaScript og kan køres i node.js runtime på en bred vifte af platforme, herunder Microsoft Windows, Google Chrome OS, Linux,

Pros:

Godt for prototyping, agile udvikling.

Hurtigt og highly scaleable.

Alt er skrevet i JavaScript

Stort ecosystem af open-source libs.

Cons:

skal ikke bruges til CPU tunge programmer.

NPM:

NPM: Node package manager ⇒ Med NPM kan man få adgang til en pakke med features som ikke er tilgængelig for JS out of the box. f.eks: "node-fetch".

Efter at have hentet node-fetch vil der blive lavet en folder node_modules. Her ligger de pakker man kan importere til brug i sit projekt.

```
import fetch from "node-fetch";
```

Explain about the Event Loop in JavaScript, including terms like; blocking, non-blocking, event loop, callback queue and "other" API's. Make sure to include why this is relevant for us as developers.

- **Blocking** : Kode der bliver kørt synchronous vil blokkere resten af programmet indtil den givne kode er eksekveret.
- **Non-blocking** : Kode som bliver kørt asynchronous venter på at callstacken bliver clear'et og bliver der efter eksekveret. På den måde undgår man at blokere andre dele af koden.
- **Call stack** : En interpreter som holder øje med hvilke kald der skal eksekveres først i en funktion. One thread = one call stack = one thing at a time. The call stack manages execution contexts and puts tasks in a queue.

Når vi i JS kalder en funktion tilføjer vores interpreter denne til vores Call stack. Hvis der er nogen funktioner inde i den givne funktion bliver disse tilføjet længere oppe i Call stacken og der efter eksekveres sekvenselt.

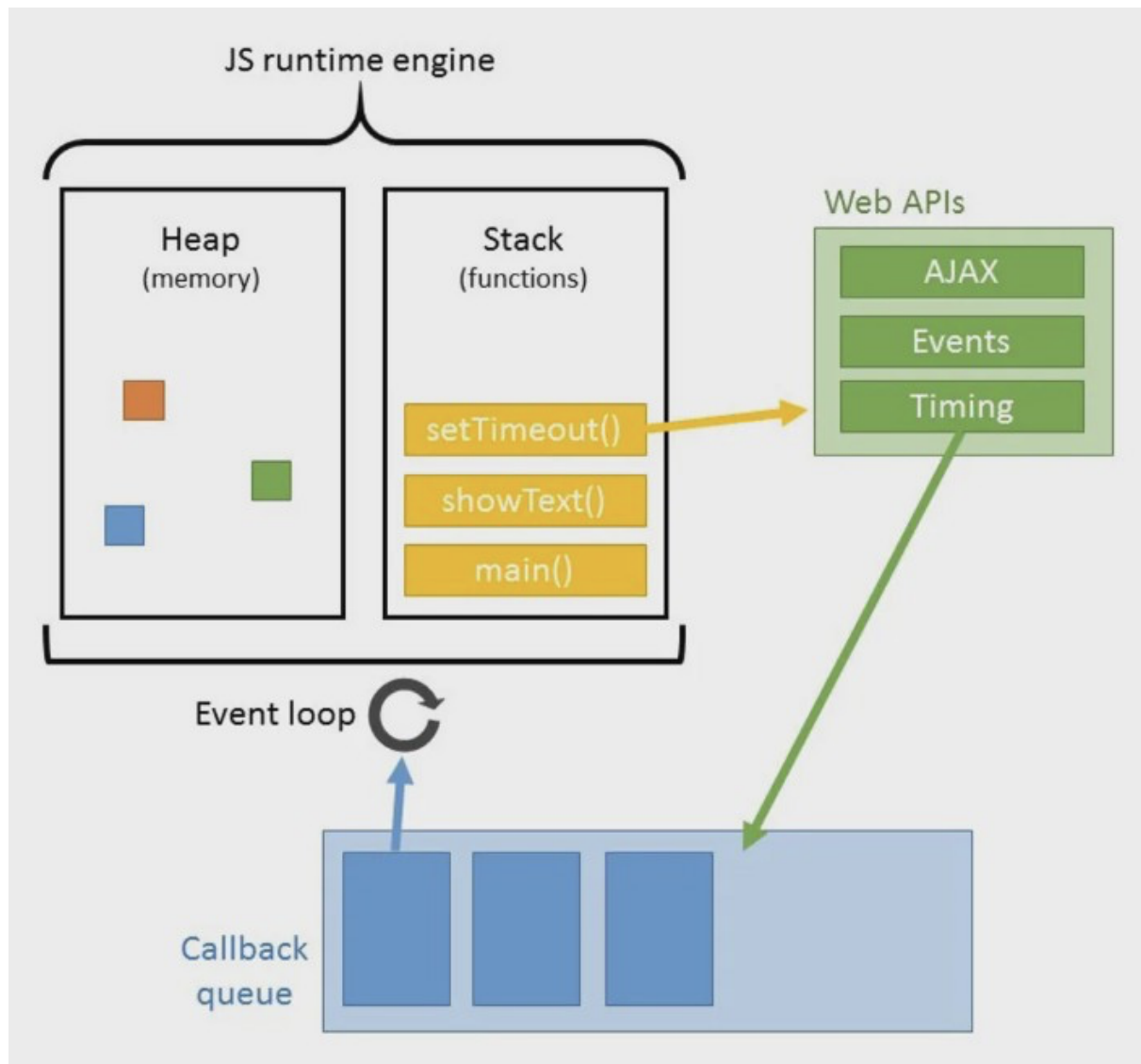
Efter funktionen er færdigt eksekveret tager interpreteren den af Call stacken igen og forsætter til næste funktion på listen.

Hvis en stack tager for meget plads kastes der en "Stack Overflow Error"

- **Event loop** : Holder øje med stack og callback queue, Hvis stacken er tom skubber den et callback ind på stack listen og derefter bliver den kodeblok

eksekveret.

Don't block the event loop → It will destroy the smoothness of a user interface.



Why is it relevant for us developers : Code runs smoother when it is non-blocking and asynchronous since the program won't freeze waiting for a task to finish.

What does it mean if a method in Node's API ends with `xxxxxxSync`?

- At metoden er synchronous
- Det er (næsten) altid bedre at køre metoder asynchronous.

- Node er async by default.

Explain the terms JavaScript Engine (name at least one) and JavaScript Runtime Environment (name at least two)

- JavaScript engine interpreter vores JS kode og konverterer det til det vi ser i browser eller node.
- **Chrome:** V8,
- **Mozilla:** SpiderMonkey,
- **Safari:** JavaScriptCore,
- **Internet Explorer:** Chakra.

Explain (some) of the purposes with the tools *Babel* and *Webpack* and how they differ from each other

Babel: Gør det nemt at arbejde med nyere versioner af JS ved at gøre det kompartibilt til ældre JS engines "browsers" . Example ES6 features til ES5.

Webpack: bundler vores projekt filer sammen så vi kan gemme dem i et module hvor den tager de dependencise med der er nødvendige for projektet.

Webpack transcriberer også nyere EcmaScript syntax til ældre JS ES kode så det er kompatible på ældre browser versioner.

Webpack.config.js fil:

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");

module.exports = {
  entry: {
    index: "./src/index.js",
    print: "./src/print.js",
  },
  devtool: "inline-source-map",
  devServer: {
    contentBase: "./dist",
  },
  plugins: [
    new CleanWebpackPlugin({ cleanStaleWebpackAssets: false }),
    new HtmlWebpackPlugin({
      title: "Output Management",
    }),
  ],
}
```

```

output: {
  filename: "[name].bundle.js",
  path: path.resolve(__dirname, "dist"),
  publicPath: "/",
},
module: {
  rules: [
    {
      test: /\.css$/,
      use: ["style-loader", "css-loader"],
    },
    {
      test: /\..(png|svg|jpg|jpeg|gif)$/i,
      type: "asset/resource",
    },
  ],
},
];
};

```

Explain using sufficient code examples the following features in JavaScript (and node)

Variable/function-Hoisting

- Hoisting: At løfte op. JavaScript rykker selv alle deklartioner op til toppen af koden. Så man kan assign en variable før man deklarerer den.

Example 1

```
x = 5; // Assign 5 to x
```

```
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x; // Display x in the element
```

```
var x; // Declare x
```

- Med dette eksempel vil vi få en undefined error da x bliver deklaret og assignet senere.

```
var x = 5; // Initialize x
```

```
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y; // Display x and y
```

```
var y = 7; // Initialize y
```

- Best practice deklare variabler i toppen for at undgå undefined errors.

***this* in JavaScript and how it differs from what we know from Java/.net.**

JavaScript *this* keyword refers til det object it belongs to.

```
var person = {  
  firstName: "John",  
  lastName : "Doe",  
  id       : 5566,  
  fullName : function() {  
    return this.firstName + " " + this.lastName;  
  }  
};
```

It has different values depending on where it is used:

- In a method, `this` refers to the **owner object**.

```
fullName : function() {  
  return this.firstName + " " + this.lastName;  
}
```

- Alone, `this` refers to the **global object**.

When used alone, the **owner** is the Global object, so `this` refers to the Global object.

In a browser window the Global object is `[object Window]`:

```
var x = this;
```

- In a function, `this` refers to the **global object**.

In a JavaScript function, the owner of the function is the **default** binding for `this`.

So, in a function, `this` refers to the Global object `[object Window]`.

```
function myFunction() {
  return this;
}
```

- In a function, in strict mode, `this` is `undefined`.

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, `this` is `undefined`.

```
"use strict";
function myFunction() {
  return this;
}
```

- In an event, `this` refers to the **element** that received the event.

```
<button onclick="this.style.display='none'">
  Click to Remove Me!
</button>
```

- Methods like `call()`, and `apply()` can refer `this` to **any object**.

In this example the person object is the "owner" of the function

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

Function Closures and the JavaScript Module Pattern

- This is called a JavaScript **closure**. It makes it possible for a function to have "**private**" variables.

```
var add = (function () {
  var counter = 0;
  return function () {counter += 1; return counter}
})();
```



```
add();  
add();  
add();
```

- The variable `add` is assigned to the return value of a self-invoking function.
- The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

A closure is a special kind of object that combines two things:

- A function
- The environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created

User-defined Callback Functions (writing functions that take a callback)

Explain the methods map, filter and reduce

Map : Tager et array og looper igennem alle elementer og eksekverer den given funtion man har tildelt map functionen. Returner et nyt array.

```
var numbers = [65, 44, 12, 4];  
var newarray = numbers.map(myFunction)  
var example2 = numbers.map(number => number * 2)  
  
function myFunction(num) {  
    return num * 10;  
}  
  
// newarray vil blive retuneret med alle tal fra numbers ganget med 10.
```

Filter : Returner et nyt array baseret på de elementer som lever op til det krav man stiller metoden.

```
var ages = [32, 33, 16, 40];  
  
function checkAdult(age) {  
    return age >= 18;  
}  
  
function myFunction() {  
    var filteredAges = ages.filter(checkAdult);  
}
```

Reduce : Reducer arrayet ned til en enkelt variable feks int.

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

Provide examples of user-defined reusable modules implemented in Node.js (learnynode - 6)

```
var fs = require("fs");
var path = require("path");

module.exports = function (dirname, ext, callback) {
  var extension = "." + ext;
  fs.readdir(dirname, function (err, files) {
    if (err) {
      callback(err, null);
    } else {
      result = [];
      files.forEach(function (entry) {
        if (path.extname(entry) === extension) {
          result.push(entry);
        }
      });
      callback(null, result);
    }
  });
};
```

```
var lslib = require("./mymodule");

var dirname = process.argv[2];
var ext = process.argv[3];

lslib(dirname, ext, function (err, files) {
  for (i = 0; i < files.length; i++) {
    console.log(files[i]);
  }
});
```

Example: Check week1 folder

Provide examples and explain the es2015 features: let, arrow functions, this, rest parameters, destructuring objects and arrays, maps/sets etc.

See her for more: <https://babeljs.io/docs/en/learn/>

Let & const : Block-scoped binding constructs. let is the new var. const is single-assignment. Static restrictions prevent use before assignment.

Arrow functions : Arrows are a function shorthand using the \Rightarrow syntax.

Unlike functions, arrows share the same lexical this as their surrounding code. If an arrow is inside another function, it shares the "arguments" variable of its parent function.

This : In JavaScript *this* keyword refers to the object it belongs to

- It has different values depending on where it is used
- In a method, this refers to the owner object.
- When used alone, the **owner** is the Global object, so this refers to the Global object.
- In a browser window the Global object is [object Window]:
- In Java, this refers to the current instance of an object which is used by a method

Rest parameters : The rest parameter syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent variadic functions in JavaScript.

```
function f(a, b, ...theArgs) {  
  // ...  
}
```

Destructuring objects and arrays : Destructuring allows binding using pattern matching, with support for matching arrays and objects.

```
const { name, realName } = hero;
```

Map : Efficient data structures for common algorithms

```
var m = new Map();  
m.set("hello", 42);  
m.set(s, 34);  
m.get(s) == 34;
```

Sets : Efficient data structures for common algorithms

```
var s = new Set();  
s.add("hello").add("goodbye").add("hello");  
s.size === 2;  
s.has("hello") === true;
```

Provide an example of ES6 inheritance and reflect over the differences between Inheritance in Java and in ES6.

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance>

```
class Person {  
  constructor(first, last, age, gender, interests) {  
    this.name = {  
      first,  
      last  
    };  
    this.age = age;  
    this.gender = gender;  
    this.interests = interests;  
  }  
  
  greeting() {  
    console.log(`Hi! I'm ${this.name.first}`);  
  };  
  
  farewell() {  
    console.log(`${this.name.first} has left the building. Bye for now!`);  
  };  
}
```

```
};
}
```

```
class Teacher extends Person {
  constructor(subject, grade) {
    this.subject = subject;
    this.grade = grade;
  }
}
```

- In ES6 there isn't much of a difference between java and javascript classes. With ES6 classes came to JS and that made JS into a OOP language like Java
- You can inherit in ES6 like in java

Explain and demonstrate, how to implement event-based code, how to emit events and how to listen for such events

The code that is being imported with has created an eventlistener with the emit function.

```
const EventEmitter = require("events");

var url = "http://mylogger.io/log";

class Logger extends EventEmitter {
  log(message) {
    console.log(message);

    // Raise an event
    this.emit("messageLogged", { id: 1, url: "http://" });
  }
}

module.exports = Logger;
```

The code that is imported and called, the eventlistener is registered with the on() method that takes a callback function:

```
const EventEmitter = require("events");

const Logger = require("../logger");
const logger = new Logger();

//register a listener
```

```
logger.on("messageLogged", (arg) => {  
  console.log("Listener called", arg);  
});
```

Example: Check week2 folder

ES6,7,8,ES-next and TypeScript

Provide examples with es-next running in a browser, using Babel and Webpack

Babel is a JS-transcompiler that can convert newer JS code to older versions.

See week 4 folder: react-webpack

Explain the two strategies for improving JavaScript: Babel and ES6 + ES-Next, versus Typescript. What does it require to use these technologies: In our backend with Node and in (many different) Browsers

ES6, ES-next, JS, Babel:

- Discussed in meetings and it takes long time for new features to be added for the following stages:
Proposal - Draft - Candidate - Finished
- The "big" guys on the market controls it
- JS is transcompiled using babel and webpack.
→ There is a barbell file in where you can setup your enviroment

```
{  
  "presets": [  
    ["env", {  
      "targets": {  
        "browsers": ["last 2 versions"]  
      }  
    }]  
  ]  
}
```

- As a user you will have to install alot of packages using NPM
→ example : npm install —save-dev babel-preset-es2015.

TypeScript

- Open source, using github, everyone can contribute
- Has to install TypeScript globally on machine or for bigger project in the project using NPM.
- TS has to be compiled to JS ⇒ You can configure a file that defines how / where the converted files should be saved.

Command : `tsc —init` ⇒ Creates the TS config file which has a lot of features. It is smart to define the output dir and set `sourceMap` to true for easier debugging.

Provide examples to demonstrate the benefits of using TypeScript, including, types, interfaces, classes and generics

Type defined variable

```
let myName: string = "Alice";
```

Class

```
class Point {  
  x: number;  
  y: number;  
}  
  
const pt = new Point();  
pt.x = 0;  
pt.y = 0;
```

Interface

```
interface IMessage {  
  original:string,  
  upperCased:string  
}
```

Generic

```
class GenericLogger<T, U> {  
  constructor() {}  
}
```

```

log = (a: T, b: U) => console.log(`Value 1: ${a}, value 2: ${b}`);
}

```

Putting it all together

```

interface IPerson {
  name: string;
}
interface IAddress {
  street: string;
}

class Person implements IPerson {
  //private _name : String;
  #name: string;
  constructor(name: string) {
    this.#name = name;
  }
  get name(): string {
    return this.#name;
  }
  set name(name: string) {
    this.#name = name;
  }
  toString(): string {
    return this.#name;
  }
}

class Address implements IAddress {
  //private _name : String;
  _street: string;
  constructor(street: string) {
    this._street = street;
  }
  get street(): string {
    return this._street;
  }
  set street(street: string) {
    this._street = street;
  }
  toString(): string {
    return this._street;
  }
}

function loggerV3(a: IPerson, b: IAddress) {
  console.log(`Person ${a.name}, address: ${b.street}`);
}

let p1 = new Person("Kurt Wonnegut");
let a1 = new Address("Lyngby Hovedgade 23");

loggerV3(p1, a1);

```



```
class GenericLogger<T, U> {
  constructor() {}
  log = (a: T, b: U) => console.log(`Value 1: ${a}, value 2: ${b}`);
}

let personlogger = new GenericLogger<IPerson, IAddress>();
personlogger.log(p1, a1);
```

Example: Check week5 folder

Explain how we can get typescript code completion for external imports.

Installing the NPM package with the @type notation will make it work for TypeScript with code completion from the IDE.

```
npm install @types/lodash
```

Explain the ECMAScript Proposal Process for how new features are added to the language (the TC39 Process)

- The process of making changes in the ECMAScript specification is done by the TC39, and naturally called the TC39 process. This process is built from five stages, starting with stage zero. Any proposal for a change in the specification goes through these stages without exception, when the committee must approve the progress from one stage to the next one.
- The process was born due to the conduct of ECMAScript 2015 edition, also known as ES6, which was a pretty huge release lasting very long without delivery (actually almost 6 years). Therefore, as of ECMAScript 2017, the editions have become smaller and are delivered every year, containing all proposals which are accepted at all stages of the process, since the previous edition.
- The TC39 process have 4 stages - Proposal - Draft - Candidate - Finished
- Link : <https://nitayneeman.com/posts/introducing-all-stages-of-the-tc39-process-in-ecmascript/>

Callbacks, Promises and async/await

Explain about (ES-6) promises in JavaScript including, the problems they solve, a quick explanation of the Promise API and:

- JavaScript is single threaded, meaning that two bits of script cannot run at the same time; they have to run one after another. A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.
- They solve the problem of only being able to run one thing at a time. A promise can be invoked and then later used, either rejected or resolved
- The Promise object works as a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
- This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.
- Promise API
 - Promise.resolve
 - The static Promise.resolve function returns a Promise that is resolved.
 - Promise.reject
 - The static Promise.reject function returns a Promise that is rejected.
 - Promise.all
 - The Promise.all() method takes an iterable of promises as an input, and returns a single Promise that resolves to an array of the results of the input promises. This returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises.
 - Promise.race
 - The Promise.race() method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

Example(s) that demonstrate how to execute asynchronous (promise-based) code in serial or parallel

- Check week 3 exercises

Example(s) that demonstrate how to implement our own promise-solutions.

- Check week 3 exercises

Example(s) that demonstrate error handling with promises

- Check week 3 exercises

Explain about JavaScripts async/await, how it relates to promises and reasons to use it compared to the plain promise API.

- Async/await får asynkronisk kode til at se ud og opfører sig som synkronisk kode.

Pros:

1. Concise and clean

Mindre kode man behøver ikke skrive .then() eller give unødvendig data med til en variable. Vi undgik også i ovenstående eksempel at nesten koden.

2. Error handling

Async/await gør det muligt at håndtere både synchronous og asynchronous errors i samme construct.

```
const makeRequest = async () => {
  try {
    // this parse may fail
    const data = JSON.parse(await getJSON())
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
```

3. Conditionals

Det er nemt at lave conditional fetches med async/await

```
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
```

```

const moreData = await makeAnotherRequest(data);
console.log(moreData)
return moreData
} else {
  console.log(data)
  return data
}
}

```

4. Intermediate values

Hvis vi har et promise 3 som første kan blive opfyldt hvis promise 1 og promise 2 bliver fufilled sammen. Så ville vi før aysnc/await benytte Array.map() og benytte promise.all() på promise 1 og 2..

Dette er meget nemmere med async/await

```

const makeRequest = async () => {
  const value1 = await promise1()
  const value2 = await promise2(value1)
  return promise3(value1, value2)
}

```

5. Error stacks

En chain fyldt med promises hvori der fejler en promise er meget nemt med asyc/await. Denne hjælper også med at debug i hvilke promise fejlen skete.

```

const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  throw new Error("oops");
}

makeRequest()
  .catch(err => {
    console.log(err);
    // output
    // Error: oops at makeRequest (index.js:7:9)
  })

```

6. Debugging

Det er ikke muligt at sætte breakpoints på arrow functions som retuner expressions (no body) man tilgængæld godt sætte breakpoints på et array kald og derfor debug som var det et almindeligt synchronous call.

```

4
5   const makeRequest = async () => {
6     await callAPromise()
7     await callAPromise()
8     await callAPromise()
9     await callAPromise()
10    await callAPromise()
11  }
12

```

Provide examples to demonstrate

- Why this often is the preferred way of handling promises
Det er syntatisk kønnere derfor mere læsbart.
- Error handling with async/await
Se ovenstående Error stacks. Eller tjek week 3 exercises.
- Serial or parallel execution with async/await.

Udtag fra week3/PromisesExercise/exercises

```

const fetch = require("node-fetch");
const URL = "https://swapi.dev/api/people/";
var now = require("performance-now");

function fetchPerson(url) {
  return fetch(url)
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      return data;
    });
}

// SEKVENSELT
async function printNamesSequential() {
  var start = now();
  console.log("Before");
  try {
    const person1 = await fetchPerson(URL + "1");
    const person2 = await fetchPerson(URL + "2");
    console.log(person1.name);
    console.log(person2.name);
  } catch (error) {
    console.error(error);
  }
}

```

```

    var end = now();
    console.log(
      "After all. Call Sequential took " +
        (end - start).toFixed(3) +
        " milliseconds. \n \n"
    );
  }

  // PARALLELT
  async function printNamesParallel() {
    var start = now();
    console.log("Before");
    try {
      const person1 = await fetchPerson(URL + "1");
      const person2 = await fetchPerson(URL + "2");
      const allResults = await Promise.all([person1, person2]);
      console.log(allResults[0].name);
      console.log(allResults[1].name);
    } catch (error) {
      console.error(error);
    }
    var end = now();
    console.log(
      "After all. Call Parallel took " +
        (end - start).toFixed(3) +
        " milliseconds. \n \n"
    );
  }

  printNamesSequential();

  printNamesParallel();

```