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 complieret 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:

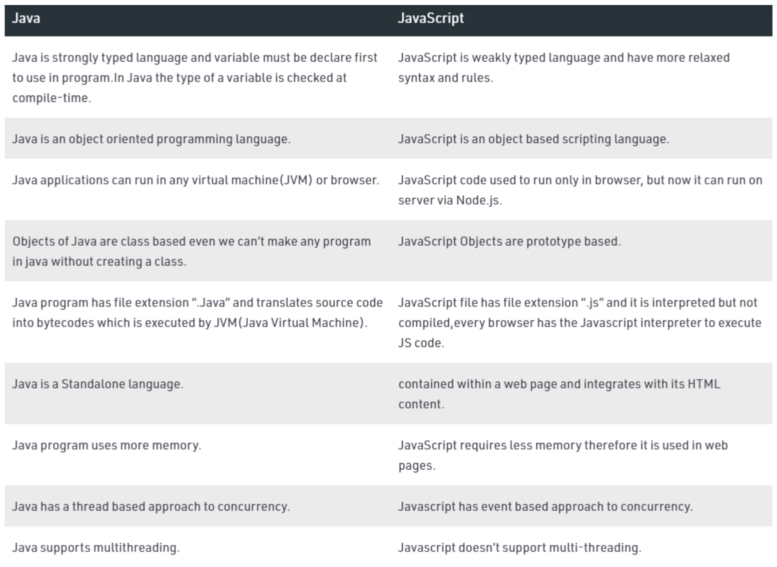
Java: er meget type stærkt og skal benytter static type checking. Dvs man skal angive hvilken type variable man giver videre feks int, string osv.

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 - Asynchrous."

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 echo system.

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 udviklig.

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 pakker med features som ikke er tilgænglig for JS out of the box. feks: "node-fetch".

Efter at have hentet node-fetch vil der blive lavet en folder node_modules. Her ligger de pakkerne man kan importer 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.

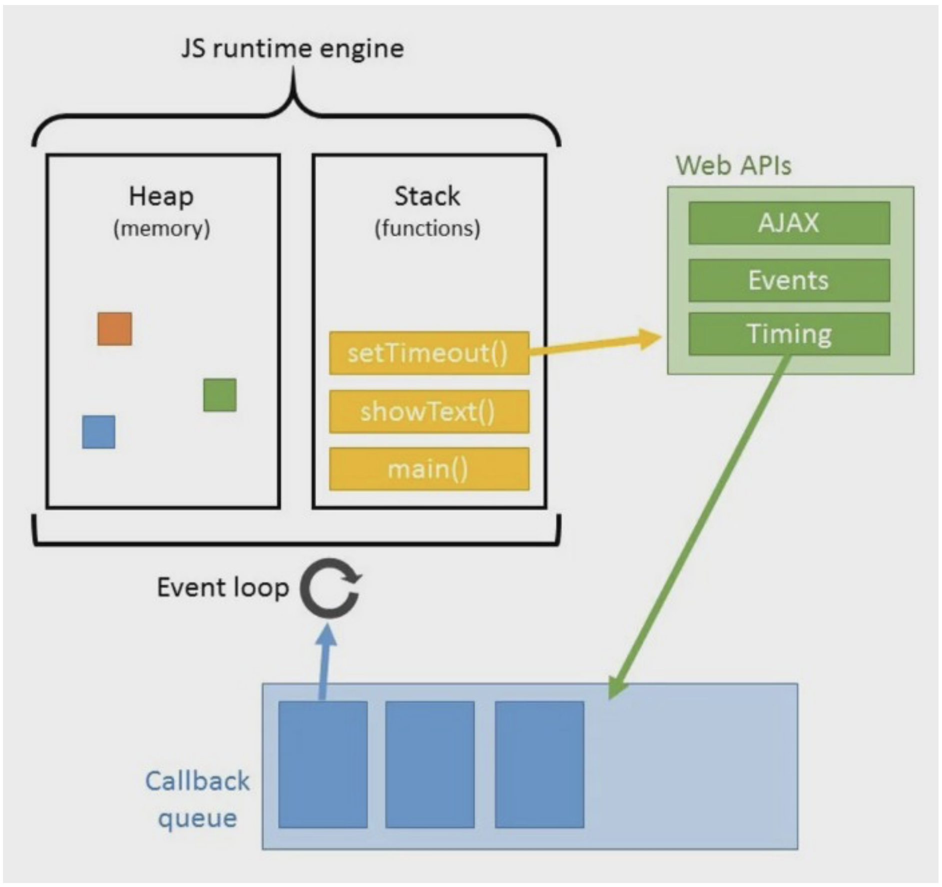
- **Blocking** : Kode der bliver kørt synchronous vil blokkere resten af programmet indtil den given kode er eksekveret.
- **Non-blocking :** Kode som bliver kørt asynchronous venter på at callstacken bliver clearet og bliver der efter eksekveret. På den måde undgår man at blokere andre dele af koden.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

  Når vi i JS kalder en funktion tilføjer vores interpreter denne til vores Call stack. Hvis der er nogen funktioner indeni i den given funktion bliver disse tilføjet længere oppe i Call stacken og der efter ekskereves 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 kastets der en "Stack Overflow Error"

- **Event loop** : Holder øje med stack og callback que, 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 wont freeze waiting for a task to finish.

What does it mean if a method in nodes API's 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 intreperter vores JS kode og konverter 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 kompertabilt 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 trascriperer også nyere EchmaScript syntax til ældre JS ES kode så det er kompletible 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 deklarer 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
```

▼ Periode 1

▼ Hoisting

💡 JavaScript rykker selv alle declarations op til toppen af koden. Så man kan assign en variable før man deklarer 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
```

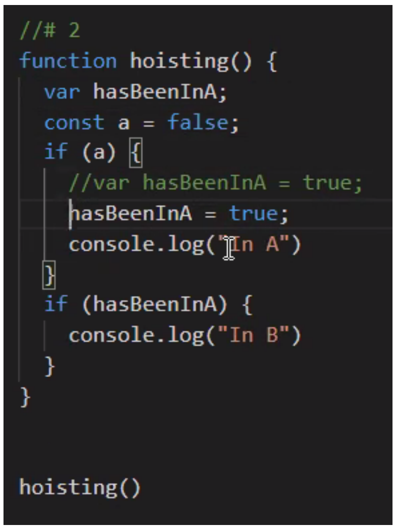
Iovenstående kan man se at X bliver først assignet, og deklareret senere.

```
Example 2

function hoisting1() {
console.log("Value of myCoolObject: " + myCoolObject);

  if(!mycoolObject) {
    var myCoolObject = {value: "Wau, I'm cool"}
    console.log(myCoolObject.value);
  }
}

hoisting1();

//OUTPUT:
//Value of myCoolObject: undefiend
//Wau, I', cool
```

Ovnestående funktion rykker altså deklartionen op til starten af funktionen, men ikke dens indhold. Derfor vil den første output være undefined og derefter vil valuen være sat inde i If statmentet.

Grunden til ovenstående er at ved brug af VAR får man et Global scope. Hvis man havde valgt at bruge **LET** istedet ville det være et **Block scope** og dermed ville d**eklarationen kun glæde indeni block koden.**

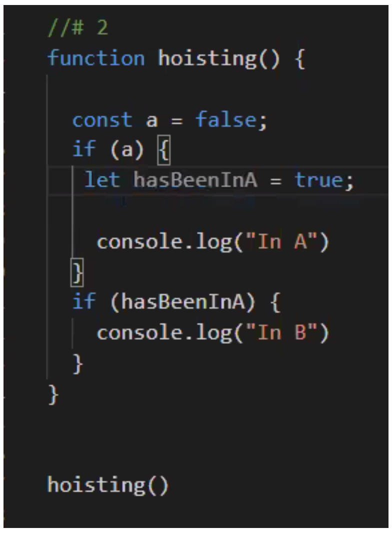
Eksempel



```
//# 2
function hoisting() {
  var hasBeenInA;
  const a = false;
  if (a) {
    //var hasBeenInA = true;
    hasBeenInA = true;
    console.log("In A")
  }
  if (hasBeenInA) {
    console.log("In B")
  }
}


hoisting()
```

Udskriver ingenting

Den udkommenteret hasBeenInA vil blive løftet op udenfor funktionen og deklareret. Den får dog først tildelt sin værdi inden i funktions blokken.



```
//# 2
function hoisting() {

  const a = false;
  if (a) {
   let hasBeenInA = true;

    console.log("In A")
  }
  if (hasBeenInA) {
    console.log("In B")
  }
}


hoisting()
```

Udskriver ingenting

Den udkommenteret hasBeenInA bliver ikke hoisted og derfor ikke løftet op. Da det er en let variable. Let bliver forbeholdt til block scopet.

▼ Video fra Lars om hoisting

  https://www.youtube.com/watch?v=8LUEREHnjOU&list=PLDbigcKhXkiUejrtlOwHBmmHxfoJ0h22g&index=1&t=160s&ab_channel=LarsMortens

▼ Closures

https://www.w3schools.com/js/js_function_closures.asp

En funktion som indenholder information om den gang den blev lavet.

Med Closures kan man lave true encaptulations.

Global Variables

A `function` can access all variables defined **inside** the function, like this:

Example

```
function myFunction() {
 var a = 4;
 return a * a;
}
```

```
// 5 Closures

function makeFunc() {
  var name = "Mozilla";

  function logName() {
    console.log(name);
  }
  return logName;
}

var f = makeFunc();
f();
```

```
function makeFunc() {
  var name = "Mozilla";

  function logName() {
    console.log(name);
  }
  function changeName(n) {
    name = n
  }
  return {
    logName,
    changeName
  };
}

var obj = makeFunc();
obj.logName()
obj.changeName("Hi Class")
obj.logName()
```

▼ This

▼ 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

Node er highly-scaleable, data-intensive og bliver brugt til real-time apps.

Node er ikke et programming language, og ikke et framework.

Pros:

Godt for prototyping, agile udviklig.

Hurtigt og highly scaleable.

Alt er skrevet i JavaScript

Stort ecosystem af open-source libs.

Cons:

skal ikke bruges til CPU tunge programmer.

Node arkitectur

JavaScript bliver interpretet af forskellige Runtime engines afhængig af hvilken browser den bliver kørt på. (Google Chrome V8, FireFox SpiderMonkey)

Tidligere kunne man kun eksekverer JS i browseren. Node gør det muligt at bruge Chromes V8 engine ved at embed det til et C++ program. Så man nu kan skrive JS lokalt på sin maskine.

Node ⇒ Runtime enviroment for JavaScript kode.

Da Runtime Envrioment er lokalt på maskinen kan vi nu benytte nye features som **fs.readFile() ⇒ Læs lokale filer på maskinen**

http.createServer() ⇒ Skab en server udfra netværket

Hvordan virker node?

Node applikationer fungerer ved begrebet **Non-blocking - Asynchrous.**

En single thread kan håndterer flere request.

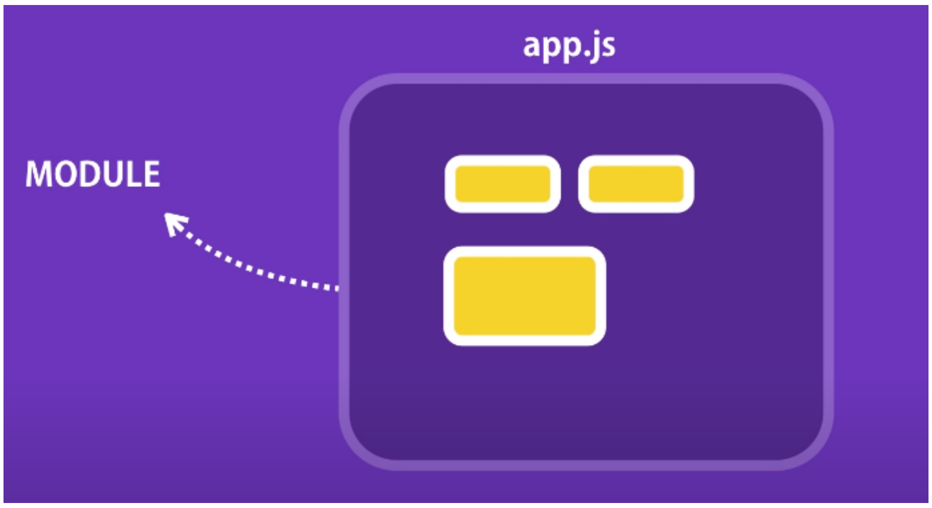
Gloabale ojects i JS

```
// Global objects i JavaScript

console.log();

setTimeout();
clearTimeout();

setInterval();
clearInterval();


// Alle objects i en browser vil have window object foran. Så window.console.log()

// Da Node køre udenfor browser står der global foran.
```

Node Module system

💡 Hver Node application har et main modul som man importerer de eksterne moduler man laver til.

I node er hvert en fil et modul og variabler og funktioner defineret i det påglædende modul er scopet til modul. Og kan derfor ikke tilgås uden for det module.



Node opdeler JS filer i modules så man skal importerer dem til hoved modulet der er app.js. På den måde undgår man fejl hvis feks 2 funktioner hedder det samme. (Tænk React)

Denne måde skaber man også private modules der kun bliver public efter de er importeret (Encapsulaction)

Understående er en indbygget modules i Node:

▼ OS

Et module til at intregerer med operativ systemet. Feks:

```
const os = require('os');

var totalMemory = os.totalmem();

console.log(`Total memory ${totalMemory}`)
```

▼ FS

Filesystem kan bruges til at interegerer med filer på maskinen lokalt.

eksempler

```
const fs = require('fs');

// Læser alle filer i directory
const files = fs.readdirSync('./')
console.log(files)

// readdir tager err og files som parameter.
// Den læser alle filer i et directory
fs.readdir('./', function(err, files) {
  if (err) console.log('error', err)
  else console.log('Result', files)
})
```

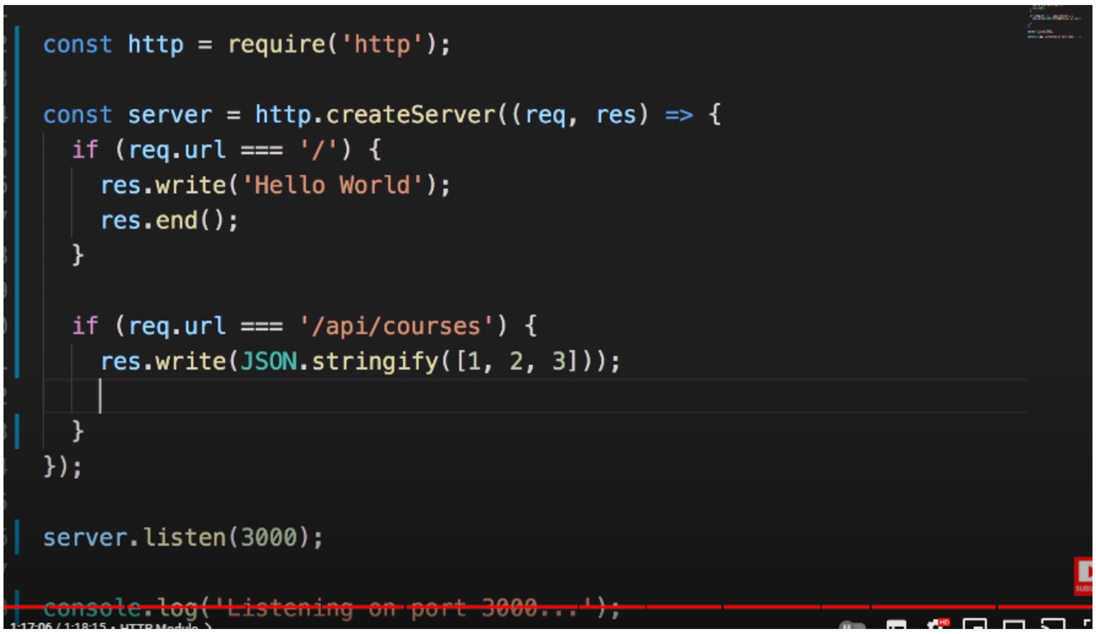
▼ Events

I node benyttet events til at lytte på hvornår der sker en forandring.

```
// For at lave en EventEmitter. Skrives med stort da det er en Class
const EventEmitter = require('event');
const emitter = new EventEmitter();

// Register en lytter. Skal lave en regiser først før emit.
// e vil være det event object vi har lavet i nedestående signal.
emitter.on('msgLogged', (e) => {
  console.log('Listener called', e)
})

// Signaler en event er sket feks msgLogged
emitter.emit('msgLogged', {id: 1 , url: 'htt://'})

// Raise: logging (data: message)
```

▼ http

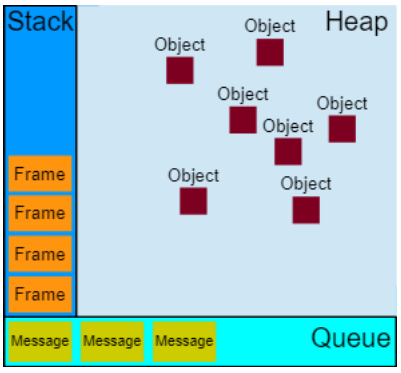
Http er et module som man kan bruge til at lave en server.



```
const http = require('http');

const server = http.createServer((req, res) => {
  if (req.url === '/') {
    res.write('Hello World');
    res.end();
  }

  if (req.url === '/api/courses') {
    res.write(JSON.stringify([1, 2, 3]));

  }
});

server.listen(3000);

console.log('Listening on port 3000...');
```

▼ Event Loop i node

> 💡 JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.



JS er et single threaded programming language dvs. det kan køre

▼ Call Stack LIFO

> 💡 Call stack er en interpreter som holder øje med hvilke kald der skal ekskeveres først i en funktion.

Når vi i JS kalder en funktion tilføjer vores interpreter denne til vores Call stack. Hvis der er nogen funktioner indeni i den given funktion bliver disse tilføjet længere oppe i Call stacken og der efter ekskereves 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 kastets der en "Stack Overflow Error"

Example

```
function greeting() {
   // [1] Some code here
   sayHi();
   // [2] Some code here
}
function sayHi() {
   return "Hi!";
}

// Invoke the `greeting` function
greeting();

// [3] Some code here
```

The code above would be executed like this:

1. Ignore all functions, until it reaches the `greeting()` function invocation.
2. Add the `greeting()` function to the call stack list.

   ```
   Call stack list:- greeting
   ```

3. Execute all lines of code inside the `greeting()` function.
4. Get to the `sayHi()` function invocation.
5. Add the `sayHi()` function to the call stack list.

   ```
   Call stack list:- sayHi- greeting
   ```

6. Execute all lines of code inside the `sayHi()` function, until reaches its end.
7. Return execution to the line that invoked `sayHi()` and continue executing the rest of the `greeting()` function.
8. Delete the `sayHi()` function from our call stack list.

   ```
   Call stack list:- greeting
   ```

9. When everything inside the `greeting()` function has been executed, return to its invoking line to continue executing the rest of the JS code.
10. Delete the `greeting()` function from the call stack list.

    ```
    Call stack list:EMPTY
    ```

In summary, then, we start with an empty Call Stack. Whenever we invoke a function, it is automatically added to the Call Stack. Once the function has executed all of its code, it is automatically removed from the Call Stack. Ultimately, the Stack is empty again.

Mulig eksamensspørgsmål:

- ▼ Explain about the Event loop in node.js (Eventloopet i JavaScript)
- ▼ Explain shortly about node.js and some of its use-cases
- ▼ Explain and demonstrate a simple application that demonstrates node ability to read OS-details, nodes Event system and a simple HTTP Server

▼ Forskel på promise, callback og async/await (CALLBACK FROM HELL)

Tjek filen filterDir og filteruser under promise excersises.

▼ Et callback from hell.

```
// callback from hell
function getPlanetforFirstSpeciesInFirstMovieForPerson(id) {
  let obj = { name: "", firstFilm: "", firstSpecies: "", homeWorld: "" };
  fetch(URLpeople + id)
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      obj.name = data.name;
      return fetch(data.films[0])
        .then((response) => {
          return response.json();
        })
        .then((data) => {
          obj.firstFilm = data.title;
          return fetch(data.species[0])
            .then((response) => {
              return response.json();
            })
            .then((data) => {
              obj.firstSpecies = data.name;
              return fetch(data.homeworld)
                .then((response) => {
                  return response.json();
                })
                .then((data) => {
                  obj.homeWorld = data.name;
                  console.log(obj);
                });
            });
        });
    });
}

// Ovenstående kan gøres meget pænere med async/await

async function getPlanetforFirstSpeciesInFirstMovieForPersonAsync(id) {
  const obj = { name: "", firstFilm: "", firstSpecies: "", homeWorld: "" };
  try {
    // På denne måde spring vi .then response over.
    const response = await fetch(URLpeople + id);
    const responseData = await response.json();
    obj.name = responseData.name;

    const response2 = await fetch(responseData.films[0]);
    const response2Data = await response2.json();
    obj.firstFilm = response2Data.title;

    const response3 = await fetch(response2Data.species[0]);
    const response3Data = await response3.json();
    obj.firstSpecies = response3Data.name;

    const response4 = await fetch(response3Data.homeworld);
    const response4Data = await response4.json();
    obj.homeWorld = response4Data.name;

    console.log(obj);
    return obj;
  } catch (error) {
    console.log(error);
  }
}
```

Uddrag fra

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/935f6051-221a-42f2-ba94-afaf7019da
49/PromisesExercise.docx
```

Execution in parallel

Fix the problem above, so that HTTP-requests are made in parallel.

Measure the time spent the same way as above, to convince yourself that;

knowing how and when to perform request in serial or parallel

is important.

Serial (sequental) kan være nyttet hvis man skal bruge data fra et respons til at lave et nyt requets. Parallel er nyttet hvis man bare skal lave en masse kald på en gang.

Tjek ex3-AsyncFuc under dag 3 Javascript 4sem. For kode eksempel.

▼ Promise

https://bitsofco.de/javascript-promises-101/

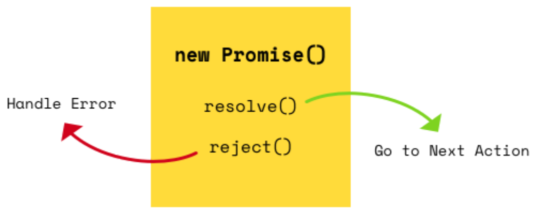
💡 JavaScript Promise represents the result of an operation that hasn't been completed yet, but will at some undetermined point in the future. An example of such an operation is a network request. When we fetch data from some source, for example an API, there is no way for us to absolutely determine when the response will be received.

Et promise kan have 3 states:

1. Pending when the final value is not available yet.
2. Fulfilled when and if the final value becomes available
3. Rejected if an error prevented the final value from being determined

for at lave et promise skal man benytte:

```
var promise = new Promise( function(resolve, reject) { /* Promise content */ } )
```

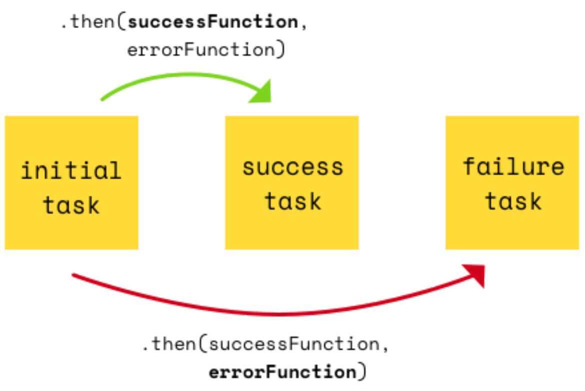


Hvis et promise bliver fulfilled kalder vi resolve()

Hvis et kald bliver rejected eller fejler kalder vi reject()

```
function get(url) {
  return new Promise(function(resolve, reject) {

    var req = new XMLHttpRequest();
    req.open('GET', url);

    req.onload = function() {
      if (req.status == 200) {
          resolve(req.response); /* PROMISE RESOLVED */
      } else {
          reject(Error(req.statusText)); /* PROMISE REJECTED */
      }
    };

    req.onerror = function() { reject(Error("Network Error")); };
    req.send();
  });
}
```

Using promises



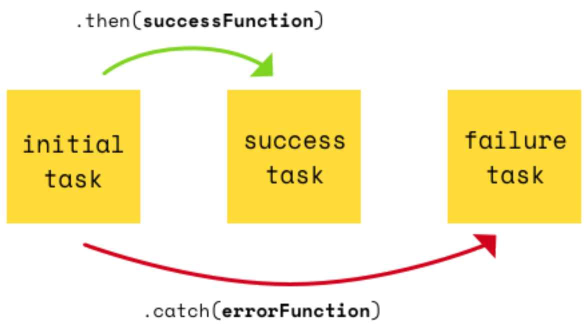
Efter at have lavet et promise kan man bruge .then() metoden.

then() vil eksekvere når et promise ikke længere er i pending state.

then() indeholder 2 optional parameters. Første parameter er til et forfuilled promise, det andet parameter er hvis det et rejected promise.

```
get(url)
.then(function(response) {
    /* successFunction */
}, function(err) {
    /* errorFunction */
})
```

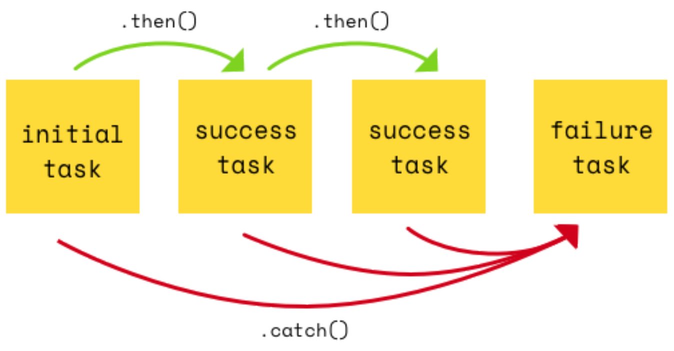
Handling Errors



For at nemmere fejl håndter kan man benytte .catch() istedet.

```
get(url)
.then(function(response) {
    /* successFunction */
})
.catch(function(err) {
    /* errorFunction */
})
```

Chaining

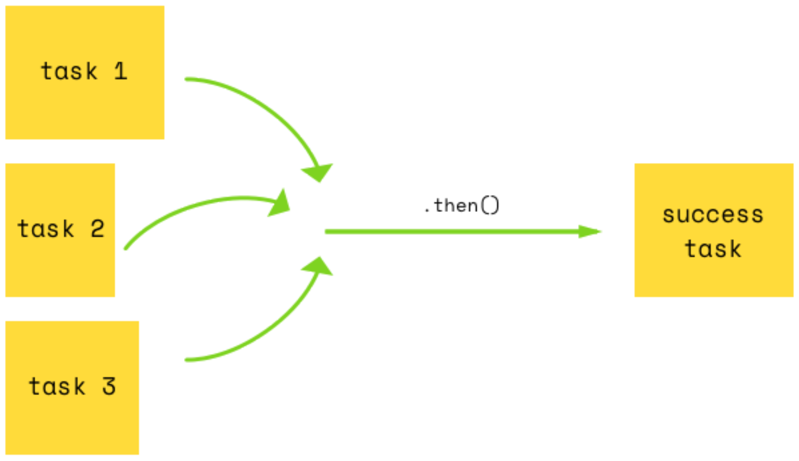


Hvis vi har en kæde af asynkoniske funktioner som vi gerne vil have eksekveret i en order kan man "chain" .then() og catch() sammen.

```
get(url)
.then(function(response) {
    response = JSON.parse(response);
    var secondURL = response.data.url
    return get( secondURL ); /* Return another Promise */
})
.then(function(response) {
    response = JSON.parse(response);
    var thirdURL = response.data.url
    return get( thirdURL ); /* Return another Promise */
})
.catch(function(err) {
    handleError(err);
});
```

Ovenstående kode viser at hvis den første .then() er fulfilled vil den rykke videre ned til den næste da den retuner et promise. Ved hvert punkt hvis den fejler vil den komme ned i catch blokken.

Executing Promises in Parallel



Hvis man har en masse promises man ønsker være fulfilled før at man køre en funktion kan man lave et Array af promises og benytte **Array.map()** til at køre et get på alle.

```
var arrayOfURLs = ['one.json', 'two.json', 'three.json', 'four.json'];
var arrayOfPromises = arrayOfURLs.map(get);

Promise.all(arrayOfPromises)
.then(function(arrayOfResults) {
    /* Do something when all Promises are resolved */
})
.catch(function(err) {
    /* Handle error is any of Promises fails */
})
```

Her benyttes Promise.all() som tager array af promises og kun hvis **ALLE** promises i arrayet bliver fulfilled vil den eksekverer funktionen. Ellers kommer den ned til catch blokken.

▼ Async/Await

https://hackernoon.com/6-reasons-why-javascripts-async-await-blows-promises-away-tutorial-c7ec10518dd9

💡 Async/await is a new way to write asynchronous code. Previous alternatives for asynchronous code are callbacks and promises.Async/await is actually just syntax sugar built on top of promises. It cannot be used with plain callbacks or node callbacks.Async/await is, like promises, non blocking.Async/await makes asynchronous code look and behave a little more like synchronous code. This is where all its power lies.

Syntax

Et eksempel på hvordan man før har brugt et promise med noget json og vil skrive "done" når man er færdig med sit request.

```
// Gamle måde med Promise
const makeRequest = () =>
  getJSON()
```

```
    .then(data => {
      console.log(data)
      return "done"
    })

makeRequest()

// Nye måde med Async/await

const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}

makeRequest()
```

await keyworded er kun tilgængligt i en funktion defineret som async. Man kan altså ikke benytte await udenfor en async funktion

```
// this will not work in top level
// await makeRequest()

// this will work
makeRequest().then((result) => {
  // do something
})
```

Pros:

1. **Cocise and clean**
   Mindre kode man behøver ikke skrive .then() eller give unødvendig data med til en variable. Vi undgik også i ovenstående eksempel at neste koden.
2. **Error handling**
   Async/await gør det muligt at håndterer både synchronous og asyncronous 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
  }
}
```

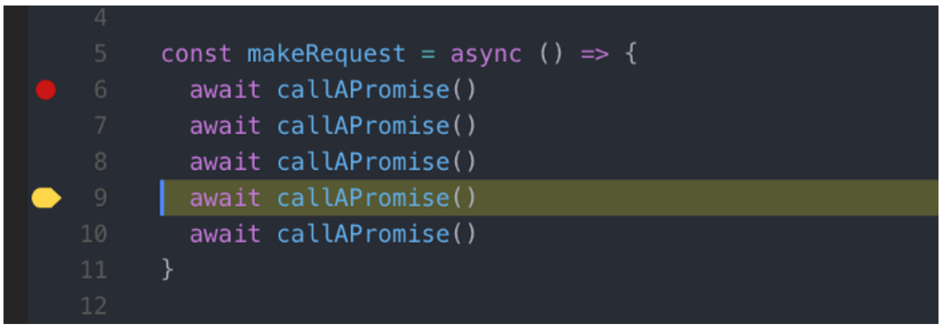
4. **Intermediate values**
   Hvis vi har et promise 3 som første kan blive opfyldt hvis promise 1 og promise 2 bliver fufilled sammen. Så ville vi før aysnc/await benytte Array.map() og benytte promise.all() på promise 1 og 2..
   Dette er meget nemmere med async/await

```
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 tilgengæld godt sætte breakpoints på et array kald og derfor debug som var det et almindeligt synchronous call.



```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
}
```

In Conclusion

Async/await is one of the most revolutionary features that have been added to JavaScript in the past few years. It makes you realize what a syntactical mess promises are, and provides an intuitive replacement.

▼ Webpack

💡 Webpack is an open-source JavaScript module bundler. It is made primarily for JavaScript, but it can transform front-end assets such as HTML, CSS, and images if the corresponding loaders are included. webpack takes modules with dependencies and generates static assets representing those modules.

https://en.wikipedia.org/wiki/Webpack

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 trascriperer også nyere EchmaScript syntax til ældre JS ES kode så det er kompletible på ældre browser versioner.

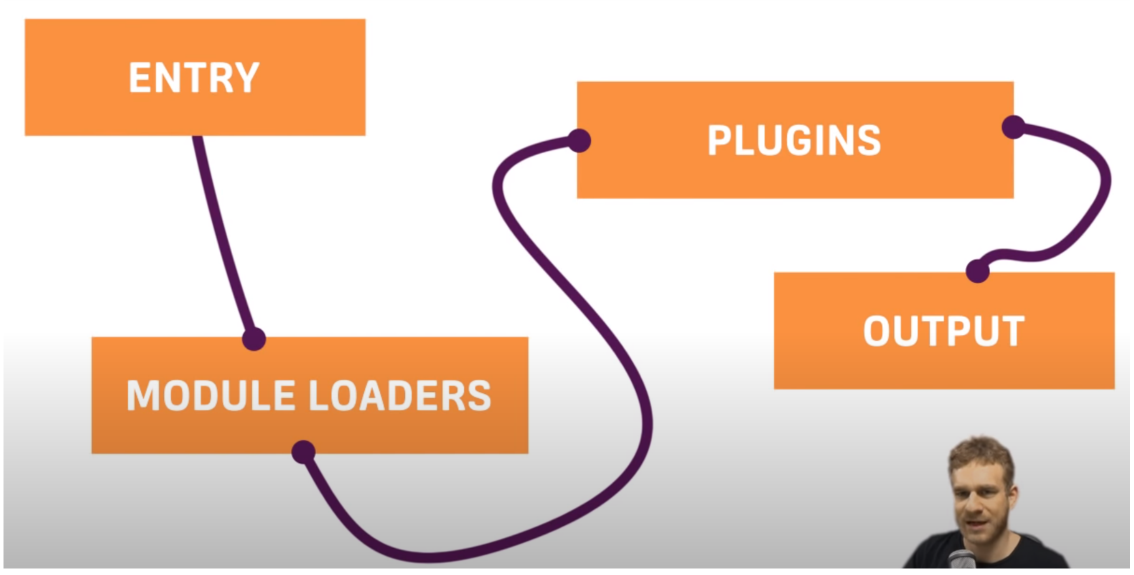
Node.js is required for using webpack.

▼ Video om webpack

https://www.youtube.com/watch?a=1&v=GU-2T7k9NfI&ab_channel=Academind

Webpack Core Concepts

1. Entry
   En webpack skal altid have et entry point hvor den skal lede efter dependencise. Den kan have flere entry points
2. Output
   Hvor skal webpacken gemme bundlen. Og hvis der er flere entry points hvor skal den gemme de forskellige bundles
3. Module loaders
   Transform our code
4. Plugins
   Gør nogenlunde det samme som modules loaders men på et andet nivaue.



▼ Babel

💡 Babel is a free and open-source JavaScript transcompiler that is mainly used to convert ECMAScript 2015+ (ES6+) code into a backwards compatible version of JavaScript that can be run by older JavaScript engines. Babel is a popular tool for using the newest features of the JavaScript programming language.

▼ Forventninger eksamen

▼ Package.json (webpack.config.js)

Scripts: som standard kan man køre npm test og start. Alle andre skal man tilføje run foran. Så eksempel:

npm start ⇒ standard

npm test ⇒ standard

npm run build ⇒ run foran

npm run etc... ⇒ run foran

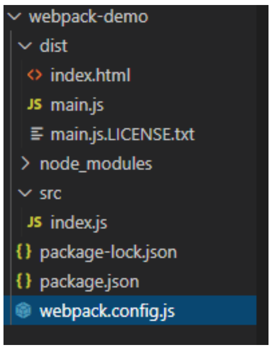
```
{
  "name": "babelex2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "babel-node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.13.1",
    "@babel/node": "^7.13.0",
    "@babel/preset-env": "^7.13.5"
  },
  "dependencies": {
    "node-fetch": "^2.6.1"
  }
}
```

devDependencies er de dependecies man benytter i development miljøet.

Dependencies bliver benyttet både i dev men også i prod miljøet.

▼ Webpack.config

Fra JavaScript Exercises (day-4)



```
webpack-demo
  dist
    index.html
    main.js
    main.js.LICENSE.txt
  node_modules
  src
    index.js
  package-lock.json
  package.json
  webpack.config.js
```

```
//Indhold: webpack.config.js

const path = require("path");

module.exports = {
  entry: "./src/index.js",
  output: {
    filename: "main.js",
    path: path.resolve(__dirname, "dist"),
  },
};
```

Webpack.config exporter module

Entry ⇒ ./src/index.js altså den javascript vi koder (læsbart)

Output ⇒ main.js under dist. Den javascript kode webpack transcripere for os. (ulæstbart)

Dette er den basic setup:

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- main.js
  |- index.html
|- /src
  |- index.js
|- /node_modules
```

▼ Asset management

▼

Nedestående installer en række dependencises og gemmer dem i -dev miljø

npm install @babel/core @babel/node @babel/preset-env --save-dev

Deployment dependencies vil blive gemt under almindelig dependencies

```
{
  "name": "babelex2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },

  "keywords": [],
  "author": "",
  "license": "ISC",
// DEV miljø
  "devDependencies": {
    "@babel/core": "^7.13.1",
    "@babel/node": "^7.13.0",
    "@babel/preset-env": "^7.13.5"
  },

// Prod miljø
  "dependencies": {
    "node-fetch": "^2.6.1"
  }
}
```

▼ Setup React without create-react-app

Følgende er hvad der er nødvendigt for at lave react app

npm init -y ⇒ laver vores package.json

npm i react react-dom ⇒ henter react og react-dom til vores dependencies som vi skal bruge i prod miljø.

npm i —save-dev webpack webpack-dev-server webpack-cli henter webpacks og gemmer dem i devDependencies da vi skal bruge dem i dev miljøet

> 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.

npm i —save-dev

- @babel/core transplier
- @babel-preset-env complier ES6 og senere versioner. Kigger på users browser
- @babel-preset-react laver JSX til vanilla JS
- babel-loader gør det muligt for babel at køre med webpack.
- html-webpack-plugin generer vores build HTML fil.
- style-loader css-loader til vores CSS filer
- file-loader til at loade filer som billeder PNG, JPG osv..

babel bruger React

React bruger ES6 (import etc..) Så vi skal bruge babel til at transpile det ned til browserfriendly code. Dette gør babel preset rea

babel loader jsx React bruge til at template

html plugin generer vores build html fil.

Webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");

module.export = {
  entry: "./src/index.js",
  output: {
    path: path.join(__dirname, "/dist"),
    filename: "index_bundle.js",
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader",
        },
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: "./src/index.html",
    }),
  ],
};
```

Hele vores React application vil være under /src hvor vores index.js ligger. Det vil blive complied bundle til output i /dist

Under module/rules laver vi test: og js. Det vil sige at alt JS skal babel complie. Exclude node_modules så node_modules ikke bliver rørt.

Use 'babel-loader' så den benytter babel-loader.

I plugins bliver der tilføjet index.html så den laver en template og gemmer i index.html

Vi laver en ny fil som hedder .babelrc

```
{
    "presets": ["env", "react"]
}
```

Så ved babel-env og babel-react at den skal bruge de presets.

Vi kan nu gå ind i vores index.js og importer React og ReactDOM da vi kan bruge ES6

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./compoennts/App";

// Vi tager elementByid på app da det er der Id ligger i vores index.html
ReactDOM.render(<App />, document.getElementById("app"));
```

I components App.js

```
import React from "react";

function App() {
  return (
    <div>
      <h1>Sup</h1>
    </div>
  );
}

export default App;
```

I package.json bliver følgende tilføjet

```
"scripts": {
    "start": "webpack-dev-server --mode development --open --hot",
    "build": "webpack --mode production"
  },
```

npm start vil starte vores app i devopment mode og —hot reloader ved hver ændring vi laver fra VSC

npm build vil 'bygge' vores projekt gennem webpack cli til produktion.

▼ TypeScript

Commands:

tsc —init

tsc ⇒ complier vores TypeScript kode til JS og ligger i vores build folder.

Npm installs:

npm install ts-node --save-dev ⇒ Compiler automatisk vores TS kode til JS. (uden vi skal skrive tsc)

💡 TypeScript er et superset af JavaScript dvs. alt man kan i JS kan man også I TS

💡 TypeScript is transpiled to JavaScript and then interpreted by a browser or node.js

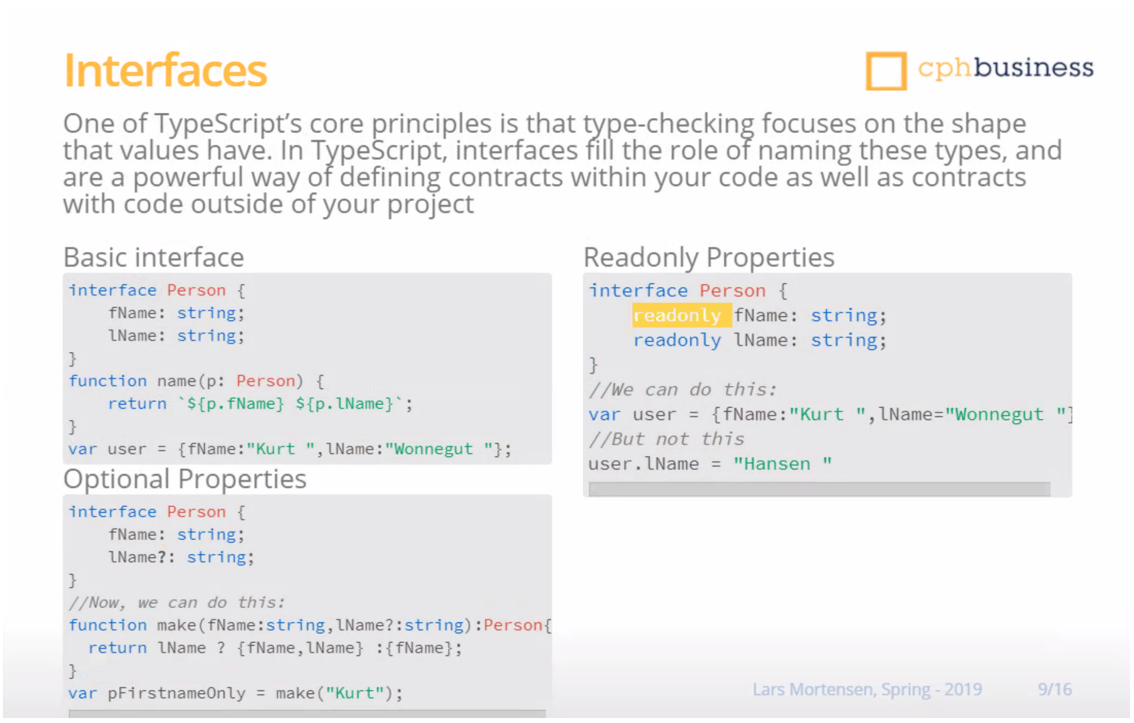
Transpilers are also called source-to-source compilers.

That means that they take a source file of language A (Typescript) and transpile it to language B (JavaScript). Both input and output languages are source files.

Compilers are only different from transpilers by the level of abstraction between input language and output language.

soruceMap gør det muligt at debug koden. Ved at den laver en map i build mappen.

Man kan lave interfaceses med TypeScript



Interfaces

cphbusiness

One of TypeScript's core principles is that type-checking focuses on the shape that values have. In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project

Basic interface

```
interface Person {
    fName: string;
    lName: string;
}
function name(p: Person) {
    return `${p.fName} ${p.lName}`;
}
var user = {fName:"Kurt ",lName:"Wonnegut "};
```

Optional Properties

```
interface Person {
    fName: string;
    lName?: string;
}
//Now, we can do this:
function make(fName:string,lName?:string):Person{
  return lName ? {fName,lName} :{fName};
}
var pFirstnameOnly = make("Kurt");
```

Readonly Properties

```
interface Person {
    readonly fName: string;
    readonly lName: string;
}
//We can do this:
var user = {fName:"Kurt ",lName="Wonnegut "}
//But not this
user.lName = "Hansen "
```

Lars Mortensen, Spring - 2019 9/16

- Med dette eksempel vil vi få en undefined error da y 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 varible 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.

Se her for mere: 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 => 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) {
  // ...
}
```

Destructuering 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 registrered 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}, adress: ${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 @type/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 exericses**

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

- **Check week 3 exericses**

Example(s) that demonstrate error handling with promises

- **Check week 3 exericses**

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

- Async/await får asynchronisk kode til at se ud og opfører sig som synchronisk kode.

Pros:

1. **Cocise and clean**
   Mindre kode man behøver ikke skrive .then() eller give unødvendig data med til en variable. Vi undgik også i ovenstående eksempel at neste koden.
2. **Error handling**
   Async/await gør det muligt at håndterer både synchronous og asyncronous 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
  }
}
```

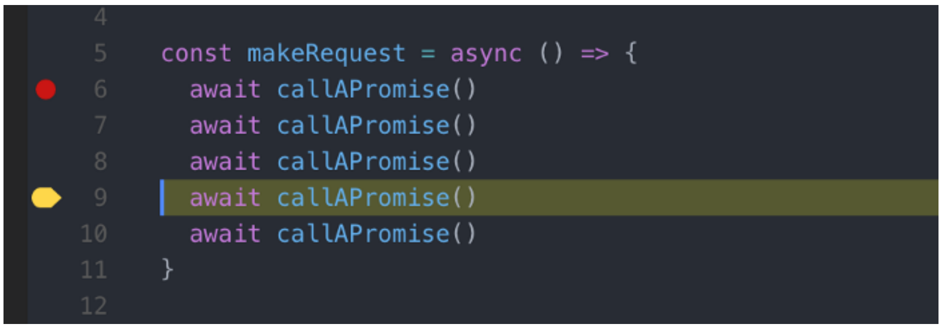
4. **Intermediate values**
   Hvis vi har et promise 3 som første kan blive opfyldt hvis promise 1 og promise 2 bliver fufilled sammen. Så ville vi før aysnc/await benytte Array.map() og benytte promise.all() på promise 1 og 2..
   Dette er meget nemmere med async/await

```
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 tilgengæld godt sætte breakpoints på et array kald og derfor debug som var det et almindeligt synchronous call.



```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
}
```

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 printNamesSequental() {
  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 Sequental took " +
      (end - start).toFixed(3) +
      " millisecounds. \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) +
      " millisecounds. \n \n"
  );
}

printNamesSequental();

printNamesParallel();
```