Periode 1

JavaScript, ES-Next, Node, Babel, Webpack, og TypeScript

Indholdsfortegnelse

JavaScript

This

Dette *keyword* er notorisk for at have flere forskellige 'definitioner' alt efter hvor det bliver brugt. I det følgende gennemgår vi kort hvordan this bliver brug i forskellige sammenhænge.

1. Brugt i forbindelse med *dot-notationen*, med et objekt på venstre side, så referer this, inden i funktionen der bliver kaldt, til objektet hvorfra metoden blev kaldt.

```
1. function City(name,country,population){
2.     this.name = name;
3.     this.country = country;
4.     this.population = population;
5.
6.     this.printCity() {
7.         console.log(this);
8.     }
9. }
10.  var c = new City("Copenhagen", "Denmark", "5mil")
11.  c.printCity()
```

   Inden i metoden *makeGesture*, vil this referere til det person objektet.

2. Brugt i en funktion, defineret i globalt scope, referer this til det globale objekt. (window/browser)
3. Brugt i en funktion, defineret i globalt scope, hvor 'strict mode' er slået til, her vil this være *undefined*
4. Ved brug af *call(), apply()*, eller *bind()* er det muligt at eksplicit definerer hvad *this* skal referer til.

```
1. class Person {
2.     constructor(name, age){
3.         this.name = name;
4.         this.age = age;
5.         this.printPerson = this.show.bind(this);
6.     }
7.
8.     printPerson = function() {
9.         console.log(this);
10.      }
11.
12.      printPersonAsync() {
13.          setTimeout(this.printPerson, 2000);
14.      }
15.  }
```

Hoisting

I JavaScript er det muligt, ved brug af *var* deklarerer variabler efter de bliver brugt, ligeledes med metoder. Dette betyder dog ikke at det er muligt at gøre brug af disse, før efter de bliver defineret. I eksemplet herunder er det muligt at se at variablen aNumber kan blive defineret som noget af de sidste, men grundet hoisting vil variablen aNumber blive trukket op til toppen af scopet, men ikke defineringen (altså = 10)

Dette gælder også for metoder, de vil også blive 'trukket' op til toppen af scopet, men definitionen vil først være på plads på den linje hvorpå dette bliver sat. Se linje 13 herunder:

```
1. if(typeof(aNumber) === 'undefined')
2.     console.log("At this point the variable aNumber is not defined, but it
    has been hoisted")
3. if(typeof(f3) === 'undefined')
4.     console.log("At this point f3 is undefined, but it has been hoisted.")
5. f1()
6. function f1(){
7.     console.log("The function f1 has been hoisted")
8.     f2()
9. }
10.  function f2() {
11.      console.log("f2 is defined below f1 but is called from within f1")
12.  }
13.  var f3 = function() {
14.      console.log("This can't be called prior to this line, because the de
   claration is only hoisted, we don't really know what f3 is prior to this p
   oint.")
15.  }
16.  var aNumber = 10
17.  console.log(`On this line aNumber has the following value: ${aNumber}`)
```

Closure

I det følgende kodeeksempel, er det muligt at se hvordan Closures, kan bruges til at holde variabler 'i live' efter scopet hvori variablen er defineret, udgår.

```
1. var makeCounter = () => {
2.     var privateCounter = 0;
3.     function changeBy(val) {
4.         privateCounter += val;
5.     }
6.     return{
7.         inc: () => {changeBy(1);},
8.         dec: () => {changeBy(-1);},
9.         value:() => {return privateCounter;},
10.      }
11.  };
```

Fordi variablen *privateCounter* bliver brugt i funktionen value, vil variablen persisterer selv udenfor dens scope. Dette betyder at variablen bliver holdt og derfra kan bruges som i det følgende eksemple.

ES6…

Helt generelt er ECMA Script en JavaScript standard, som tilføjer en masse simplificeringer og funktionalitet til JavaScript, heriblandt *let, arrow-functions, class,* for blot at nævne et par.

Promise

Promises er et lidt underligt element i JavaScript, i form af at Promises et løfte om noget der vil blive returneret på ét eller andet tidspunkt. Dette giver ikke så meget mening når man tænker på at JavaScript er et enkelt-trådet programmeringssprog og derfor ikke selv kan klare parallelisering. Det kan browseren, eller node, dog i stedet og derved kan vi godt arbejde med parallelisering.

Et promise er, som sagt, et løfte om noget, der returneres på et *ubestemt* tidspunkt. Det der bliver returneret, kommer enten tilbage i form af et *resolve* eller et *rejekt*. Når man danner et promise så specificerer man hvad der er der bliver sendt tilbage når alt går godt, i dette *resolve*. Hvis der er noget der ikke går godt, så bliver dette returneret i *reject.* Den følgende kode er et eksempel på hvordan *resolve* og *reject* kan defineres:

```
1. var myPromise = (size) => new Promise(function(resolve,reject) {
2.     crypto.randomBytes(size, function(err,buffer) {
3.         if(err) {
4.             reject(err)
5.         } else {
6.             resolve({
7.             "length": size,
8.             "random": buffer.toString('hex')
9.             })
10.          }
11.      })
12.  });
```

På linje 3 kan vi se at hvis der sker en fejl i metoden randomByte, så afviser vi dette *promise*, og sender fejlbeskeden tilbage. Hvorimod hvis der ikke er noget der går galt, så går vi videre til linje 6 hvor vi i vores *resolve* danner et objekt som vi gerne vil returnere.

Async / Await

Hvor promises er en genial metode til at håndtere hændelser der gerne skulle ske asynkront og helst ikke skulle stå og blokerer den eneste tråd vi har at arbejde med. Så er der tilfælde hvor vi meget gerne vil have et resultet tilbage før vi vælger at gå videre med arbejdet. Dette er dog muligt i Promises ved brug af then()-chanining. Men skulle det ske at vi skal vente på noget bestemt så er det også mulig t at gøre brug af Await i en asynkron metode. Det følgende kodeeksempel viser hvordan vi kan lave et fetch kald, hvor vi derefter afventer des resultat, før vi går videre med det næste der skal eksekveres.

```
1. async function fetchPerson(url){
2.     let response = await fetch(url);
3.     return await response.json();
4. }
```

I dette lille stykke kode, kan vi se at vi kalder fetch med await foran, dette betyder at vi står og venter på at få responsen tilbage fra dette kald. Ligeledes når vi så kalder .json() metoden så venter vi altså også med at returnerer dette til at der bliver givet et resultat fra denne metode.

Let

Keyword'et let er noget som vi kender fra andre programmeringssprog, dette bruges nemlig til at danne en variabel der er bundet til sit lokale scope. Vi ved at keyword'ët var bliver hoistet op, og kan derfor bruges udenfor et lokalt scope, dette sker ikke med let, bliver let defineret inden i et lokalt scope, så kan dette kun

bruges inden i det lokale scope hvor den blev defineret. Dette er som vi kender det fra f.eks. Java hvor alle variabler er lokalt scoped.

Arrow function

Med *arrow-functions*, kan vi skrive funktioner på en ny, mere læsbar, måde. I den følgende kode kan det ses hvordan det er muligt at gøre brug af denne nye måde at skrive funktioner på.

```
1. // Regular Function
2. hello = function() {
     return "Hello World!";
   }
```

```
1. // Arrow Function
2. hello = () => {
     return "Hello World!";
   }
```

Dette har dog også en betydning for *this*, hvor i en almindelig funktion *this* ville referere til objektet der kaldte funktionen, så vil *this* i en arrow-function altid referer til det objekt der definerede den.

Class & Inheritance

I ES6 blev det muligt at gøre brug af en speciel type af funktion, class. Dette betyder at vi kan begynde at skrive 'klasser' lidt ligesom vi kender det fra Java og andre objekt-orienterede programmeringssprog.

Da klasser blot er en anden type funktioner, så skriver vi følgende kode for at definerer en 'klasse' i JS (ES6). Her kan vi se at for at definerer vores klasses attributter, så skal disse defineres inden i klassens konstruktør.

```
1. class Car {
     constructor(brand) {
       this.carname = brand;
     }
   }
```

Ligesom vi kender det fra Java, så har vi også mulighed for at gøre brug af nedarvning. Ved brug af samme klassse som i overstående eksempel har vi mulighed for at implementerer nedarvning på følgende måde:

```
1. class Model extends Car {
     constructor(brand, mod) {
       super(brand);
       this.model = mod;
     }
     show() {
       return this.present() + ', it is a ' + this.model;
     }
   }
```

Node

Hvor almindeligt JavaScript kode oftest bliver brug til at skrive klient-side kode, til brug i en browser, så er det muligt, ved brug af Node.js, at skrive backend-kode. Node.js er altså et runtime environment, der gør

det muligt at køre javascript kode udenfor en browser. Og det er dermed muligt at bruge javascript til fullstack udvikling, altså både frontend og backend. Ligesom en browser giver JavaScript muligheden for at eksekverer kode asynkront så leverer Node.js også denne funktionalitet.

NPM

NPM er Nodes package manager, da der findes mange forskellige pakker til at udvide funktionaliteten af Node, så brugs der ofte en package manager til at holde styr på disse. Dette betyder at ønskes der funktioner som fetch, kendt fra JavaScript i en browser, så er dette også muligt at tilgå, dog skal dette installeres gennem denne manager. Installation igennem NPM kunne dog ikke være nemmere da det blot kræver en enkelt linje for at hente dette ned.

```
1. npm i node-fetch
```

Event based code

I Node er det muligt at skabe kode der reagerer på bestemte hændelser, f.eks. at en bestemt forudsætning bliver opfyldt, og derfra alle lyttere bliver notificeret omkring dette og derfra kan arbejde. I det følgende kode er det muligt at se hvordan en klasse, som nedarver fra EventEmitter, kan tilkytte sig et bestemt event og når dette så forekommer udføre en bestemt handling.

```
1. class DOS_Detector extends EventEmitter {
2.     constructor(timeValue){
3.         super();
4.         this.urls = new Map();
5.         this.TIME_BETWEEN_CALLS = timeValue;
6.     }
7.     addUrl = (url) => {
8.         const time = new Date().getTime()
9.         if(this.urls.has(url)){
10.             const deltaTime = time - this.urls.get(url)
11.             if(deltaTime < this.TIME_BETWEEN_CALLS){
12.                 this.emit("DosDetected", {
13.                     url:url,
14.                     deltaTime: deltaTime
15.                 })
16.             }
17.         }
18.         this.urls.set(url,time);
19.     }
20. }
21.
22. const DOS_Detector = require('./dosDetector')
23. const detector = new DOS_Detector(3000);
24. detector.on('DosDetected', ({url, deltaTime}) => {
25.     console.log(`DosDetected!! -
    > url: ${url} - Time Between: ${deltaTime}`)
26. })
27.
```

I dette tilfælde bliver DOS_Detector'ren detector, sat til at lytte efter hændelsen 'DosDetected' og skulle dette forekomme, så skrive en besked ud til console.log. Dette ses i linje 24 og i funktionen på linje 7 er det muligt at se hvordan DOS_Detector bliver sat op til at smide en hændelse afsted.

Babel

Babel er en JavaScript tanscompiler der kan tage imod det nyeste af nyeste JavaScript kode (ES6+) og compile det ned til JavaScript der er kompatible med ældre engines. F.eks. kan babel tage arrow-functions og transformere dette om til almindelige funktioner der kan bruges på alle versioner af JavaScript engines, et eksemple på dette kunne være den følgende kode

```
1. // Babel Input: ES6 arrow function
2. [1, 2, 3].map((n) => n + 1);
3.
4. // Babel Output: ES5 equivalent
5. [1, 2, 3].map(function(n) {
6.    return n + 1;
7. });
```

Webpack

Webpack er en manager af alt din kode, styling, og assets. Dette betyder at med Webpack opsat, så kan Webpack håndtere dine javascript file, html filer, csss filer og så videre, samt pakke disse en pakker der er bedre for en webbrowser at håndterer, f.eks. ved at samle al JavaScript kode en en bundle og derefter minify disse – så de ikke fylder nær så meget og derved er hurtigere at overføre til klienten.

Herunder findes en Webpack konfigurationsfil, som er med til at definerer hvad der skal ske med diverse typer af filer, blandt andet css filer, i form af loaders. Derudover er det også denne vej igennem det er muligt at definere hvordan dine JS-filer skal pakkes, i dette tilfælde bliver filerne pakke i to pakker, henholdsvis *app* og *print*

```
1. const path = require('path');
2. const HtmlWebpackPlugin = require('html-webpack-plugin');
3. const { CleanWebpackPlugin } = require('clean-webpack-plugin');
4. module.exports = {
5.     entry: {
6.         app: './src/index.js',
7.         print: './src/print.js',
8.     },
9.     devtool: 'inline-sourde-map',
10.      devServer: {
11.          contentBase: './dist',
12.      },
13.      plugins: [
14.          new CleanWebpackPlugin({
15.              cleanStaleWebpackAssets: false
16.          }),
17.          new HtmlWebpackPlugin({
18.              title: 'Output Management',
19.          }),
20.      ],
21.      output: {
22.          filename: '[name].bundle.js',
23.          path: path.resolve(__dirname, 'dist'),
24.      },
25.      mode: 'development',
26.      module: {
27.          rules: [
28.              {
```

```
                    test: /\.css$/,
                    use: [
                        'style-loader',
                        'css-loader',
                    ],
                },
                {
                    test: /\.(png|svg|jpg|gif)$/,
                    use: [
                        'file-loader',
                    ],
                },
            ],
        },
    };
```

TypeScript

Hvor det ikke er muligt at fange mange type-relaterede fejl i JS, da dette først fanges i runtime, så er dette ikke længere et problem i TypeScript som nemt fanger disse fejl allerede ved compiletime. TypeScript giver nemlig muligheden for at definere sine variablers typer, samt funktions-argument typer. I det følgende kan vi se hvordan vi kan definere typer for variabler samt hvordan vi kan definere klasser og nedarvning, næsten som vi kender det fra Java eller lignende objekt-orienterede programmeringssprog.

```
abstract class Shape{
    private _color:string;
    constructor(color:string) {
        this._color = color;
    }
    abstract get area():number;
    abstract get perimeter():number;

    get color(): string {return this._color;}
     set color(color:string) {this._color = color;}

     toString(): string{
         return `Shape's color: ${this.color}, Area: ${this.area}, Perime
  ter: ${this.perimeter}`
     }
}
```

I det overstående stykke kode kan vi se at hver variable er blevet tilknyttet en type ved brug af : notationen. Dette gælder også for vores funktioner hvori argumenter og returntyper er defineret på samme måde. Som vi også ser her, er det også muligt at folde ud for objekt-orienteret programmering med arvehierarki der også består af abstrakte klasser der ikke direkte kan implementeres. I koden herunder ser vi hvordan vi kan gøre brug af en abstrakt klasse til nedarvning og metode *overloading*,

```
class Circle extends Shape{
    protected _radius:number;
    constructor(radius:number,color:string) {
        super(color);
        this._radius = radius;
    }
    get radius(): number {return this._radius;}
```

```
8.      set radius(radius:number) {this._radius = radius;}
9.
10.       get area(): number {
11.           return (Math.PI * Math.pow(this._radius,2));
12.       }
13.       get perimeter(): number {
14.           return (2 * Math.PI * this._radius);
15.       }
16.   }
```