ExamQuestions 5

## 2) Provide examples and explain the es2005 features: let, arrow functions, this, rest parameters, de-structuring assignments, maps/sets etc.

*Let*: let er hvor man begrænser scopet. Dvs. laver man en *let* *tekst = ” ”;* inde i en funktion så er den variabel kun tilgængelig inde i den funktion. Andre funktioner kan altså ikke se variablen.

let ***helloworld*** = **"hello world"**;  
function *log*()  
{  
 let yolo = ***helloworld***;  
 **console**.log(***helloworld***) // helloworld er declareret i scopet (es6.js) og er globalt.  
}  
*log*();  
**console**.log(yolo); // virker ikke da let kun findes i scopet (functionen log())

### Provide examples and explain the es2005 features: let, arrow functions, this, rest parameters, de-structuring assignments, maps/sets etc.

*Arrow Functions*: En Arrow Funktion har kortere syntax end en normal funktion, plus den behandler ”this” på en anden måde. Dvs hvis man laver en funktion som nedenfor så virker this som forventet.

var ***b*** = 23;  
var ***stef*** = (*x* => {  
 this.**b** = *x* + 5;  
 return this.***b***;  
})  
  
**console**.log(***stef***(5))

*An* ***arrow function expression*** *has a shorter syntax compared to* [*function expressions*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) *and does not bind its own*[*this*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this)*,*[*arguments*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments)*,*[*super*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super)*, or [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target). Arrow functions are always* [*anonymous*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name)*. These function expressions are best suited for non-method functions and they can not be used as constructors.*

Rest parameters:[Rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) allow your functions to have variable number of arguments without using the [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) object. The rest parameter is an instance of Array so all the array methods just work.

Note that there can only be a single rest parameter for a given function. This means that listAnimals(...cats, ...dogs) will not work, whilst listAnimals(...cats) will work just fine.

function *f*(*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*) {  
  
  
 let message2 = **`Sum** ${*a*+*b*}**,  
rest value 1 is a:** ${*c*.**constructor**.**name**} **rest value 1 is a:** ${*d*.**constructor**.**name**} **rest value 1 is a:** ${*e*.**constructor**.**name**} **rest value 1 is a:** ${*f*.**constructor**.**name**} **rest value 1 is a:** ${*g*.**constructor**.**name**} **rest value 1 is a:** ${*h*.**constructor**.**name**}  
**`** return message2;  
  
}  
  
**console**.log(**""**)  
**console**.log(**""**)  
**console**.log(**"Opgave 5"**)  
**console**.log(**"B) Test the rest operator using the code below"**)  
  
  
var ***rest*** = [true,2,**"hello World"**,[1,2,3],new Date(),{}];  
var ***restParams*** = [...***rest***];  
  
**console**.log(*f*(5,2,...***restParams***));

### De-structuring assignments:

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to extract data from arrays or objects into distinct variables.

var ***b*** = [6, 7, 8, 9]  
var ***a*** = [***b***, 1, 2, 3, 4, 5];  
  
[g, x, ...rest] = ***a***;  
**console**.log(g) // [6, 7, 8, 9]  
**console**.log(x) // 1  
**console**.log(rest); // [1,2,3,4,5]

### Map & sets:

In Map you can use any value as Key (even an object).   
In Sets all entries are unique. Sets are unordered.

let ***map*** = new Map();  
  
***map***.set(**'foo'**, 123);  
**console**.log(**"get the value of 'foo' in map: "** + ***map***.get(**'foo'**))  
  
  
**console**.log(**"'foo' exists in the map: "** + ***map***.has(**'foo'**))  
  
**console**.log(**"deleted 'foo' key: "** + ***map***.delete(**'foo'**))  
  
**console**.log(**"'foo' is indeed deleted: "** + ***map***.has(**'foo'**))

## 3) Explain and demonstrate how es2015 supports modules (import and export) similar to what is offered by NodeJS.

In ES2015/ES6 you can make Named Exports which are several per module, or you can export the whole module, OR you can make a default export (one per module).

In Node.js you export and import like so (see beneath) for example, and you can actually do mutableValue++ (which you cannot do in es6).

var mutableValue = 3;  
function *incMutableValue*() {  
 mutableValue++;  
}  
module.**exports** = {  
 **mutableValue**: mutableValue, // (A)  
 incMutableValue: *incMutableValue*,  
};  
  
//--------------------Different Module-----------------------------------  
  
  
var mutableValue = require(**'./examQuestions'**).**mutableValue**; // (B)  
var incMutableValue = require(**'./examQuestions'**).incMutableValue;  
// The imported value is a (disconnected) copy of a copy  
**console**.log(mutableValue); // 3  
incMutableValue();  
**console**.log(mutableValue); // 3  
// The imported value can be changed  
mutableValue++;  
**console**.log(mutableValue); // 4

and in es6 you can export and import like so (see beneath) for example. ES6 is READ-ONLY so you can call a function and get the “live value”.

export function *square*(*x*) {  
 return *x* \* *x*;  
}  
  
export function *diag*(*x*, *y*) {  
 return *x* + *y*;  
}  
  
//------------------------Different Module--------------------------  
  
  
import { square, diag } from **'lib'**;  
**console**.log(square(10)); // 100  
**console**.log(diag(4, 3)); // 7  
  
  
//You can also import the complete module:  
  
  
import \* as lib from **'lib'**;  
**console**.log(lib.square(11)); // 121  
**console**.log(lib.diag(4, 3)); // 5

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

I ES6 behøver den klasse som arver fra en anden klasse, ikke en kontstruktor. Firkant har stadig adgang til konstruktoren fra Shape. Men i javascript SKAL der være en konstruktor.

I JAVA behøves der ikke en konstruktor, men har man en konstruktor SKAL der være konstruktor i de klasser som arver.

class Shape {  
 constructor(*color*){  
 this.**\_color** = *color*;  
 this.**\_perim** = **""**;  
 this.**\_area** = **""**;  
 }  
  
}  
  
class Firkant extends Shape {  
  
 returncolor()  
 {  
 return this.**\_color**;  
 }  
  
}  
  
var ***firkant*** = new Firkant(**"grøn"**);  
// når man laver en firkant har den automatisk color,// perim og area da den arver dette fra shape  
  
**console**.log(***firkant***.returncolor())

## 5) Explain about Generators and how to use them to:

## Implement iterables

## Implement blocking with asynchronous calls

Generators are functions that can be paused and resumed, which enables a variety of applications.

Generators can play three roles:

Iterators (data producers): Each yield can return a value via next(), which means that generators can produce sequences of values via loops and recursion. Due to generator objects implementing the interface Iterable [5], these sequences can be processed by any ECMAScript 6 construct that supports iterables. Two examples are: for-of loops and the spread operator (...).

Observers (data consumers): yield can also receive a value from next() (via a parameter). That means that generators become data consumers that pause until a new value is pushed into them via next().

Coroutines (data producers and consumers): Given that generators are pausable and can be both data producers and data consumers, not much work is needed to turn them into coroutines (cooperatively multitasked tasks).

### Blocking on asynchronous function calls

In the following code, I use [the control flow library co](https://github.com/tj/co) to asynchronously retrieve two JSON files. Note how, in line (A), execution blocks (waits) until the result of Promise.all() is ready. That means that the code looks synchronous while performing asynchronous operations.

co(function\* () {  
 try {  
 let [croftStr, bondStr] = yield Promise.all([ // (A)  
 getFile(**'http://localhost:8000/croft.json'**),  
 getFile(**'http://localhost:8000/bond.json'**),  
 ]);  
 let croftJson = **JSON**.parse(croftStr);  
 let bondJson = **JSON**.parse(bondStr);  
  
 **console**.log(croftJson);  
 **console**.log(bondJson);  
 } catch (*e*) {  
 **console**.log(**'Failure to read: '** + *e*);  
 }  
});

As generator objects are iterable, ES6 language constructs that support iterables can be applied to them. The following three ones are especially important.

First, the for-of loop:

for (let ***x*** of genFunc()) {  
 **console**.log(***x***);  
}  
// Output:  
// a  
// b

Second, the spread operator (...), which turns iterated sequences into elements of an array (consult [[7]](http://www.2ality.com/2015/03/es6-generators.html#[7]) for more information on this operator):

let ***arr*** = [...genFunc()]; // ['a', 'b']

Third, destructuring [[6]](http://www.2ality.com/2015/03/es6-generators.html#[6]):

> let [***x***, ***y***] = genFunc();  
> ***x*'a'**> ***y*'b'**

## 6) Explain about promises in ES 6, Typescript, AngularJS including:

## The problems they solve.

## Examples to demonstrate how to avoid the "Pyramid of Doom"

## Examples to demonstrate how to execute asynchronous code in serial or parallel

Hvis vi gerne vil lave en async task (kan være crypto, kunne være http kald, kunne være hvad som helst) kan vi i ES2015 gøre brug af Promises. En Promise er en 'placeholder' for et fremtidigt svar/objekt, eller hvad end man vil kalde det. Promises kan gøre brug af deres .then() metode, som bliver kørt, når en Promise har fået svar fra det den bedte om (kunne være et REST-kald eller andet, som er async). Det gode ved .then() er, at det også returnerer en Promise og vi kan derfor chaine vores kald, hvis vi vil have svarene i en bestemt rækkefølge. Først når vi har fået et bestemt svar kalder vi .then() hvori vi kalder på et nyt REST-kald osv. Det fede er, at man kan slutte med en .catch() og kun skrive sin error-handling et sted. Man kunne dog også kalde en .catch efter et specifikt .then() hvis man ville. Hvis man i stedet vil lave 100 random forespørgelser og er ligeglad med rækkefølgen kan man gøre brug af Promise.all, der ikke holder styr på rækkefølgen på ens promises, men venter på at alle Promises er fuldført, og derefter kan man kalde .then() metoden, og gøre noget med sine svar.

// function somewhere in father-controller.js  
var *makePromiseWithSon* = function() {  
 // This service's function returns a promise, but we'll deal with that shortly  
 SonService.getWeather()  
 // then() called when son gets back  
 .then(function(*data*) {  
 // promise fulfilled  
 if (*data*.forecast===**'good'**) {  
 prepareFishingTrip();  
 } else {  
 prepareSundayRoastDinner();  
 }  
 }, function(*error*) {  
 // promise rejected, could log the error with: console.log('error', error);  
 prepareSundayRoastDinner();  
 });  
};

### Promises and avoiding the Pyramid of Doom:

A little preview of how a promise-based approach solves the callback hell:

// Callback approach  
async1(function(){  
 async2(function(){  
 async3(function(){  
 ....  
 });  
 });  
});  
// Promise approach  
var task1 = async1();  
var task2 = task1.then(async2);  
var task3 = task2.then(async3);  
task3.catch(function(){  
 // Solve your thrown errors from task1, task2, task3 here  
})  
// Promise approach with chaining  
async1(function(){..})  
 .then(async2)  
 .then(async3)  
 .catch(function(){  
 // Solve your thrown errors here  
 })

Let's break it down on what Promise is doing for us here:

* Flattened callbacks
* Return values from asynchronous function
* Throw and Catch exceptions

### Examples to demonstrate how to execute asynchronous code in serial or parallel:

If you chain asynchronous function calls via then(), they are executed sequentially, one at a time:

asyncFunc1()  
 .then(() => asyncFunc2());

If you don’t do that and call all of them immediately, they are basically executed in parallel (a fork in Unix process terminology):

asyncFunc1();  
asyncFunc2();

Promise.all() enables you to be notified once all results are in (a join in Unix process terminology). Its input is an Array of Promises, its output a single Promise that is fulfilled with an Array of the results.

Promise.all([  
 asyncFunc1(),  
 asyncFunc2(),  
])  
 .then(([*result1*, *result2*]) => {  
 •••  
 })  
 .catch(*err* => {  
 // Receives first rejection among the Promises  
 •••  
 });

## 7) Explain about TypeScript, how it relates to JavaScript, the major features it offers and what it takes to develop Server and Client side applications with this technology.

Typescript er bygget på javascript: The TypeScript compiler is itself written in TypeScript, transcompiled to JavaScript. Det er class-based.

En af de større ting er at man kan lave types:

*let isDone: boolean = false;*

*let msg: string = "Hello World";*

*//Array*

*let numbers: Array<number> = [1,2,3];*

*//Enum*

*enum Color {Red, Green, Blue}*

*//void*

*function warnUser(): void {*

*alert("This is my warning message");*

*}*

TypeScript add Classes and Inheritance "much" like we know from Java/C# with:

Class, Constructor, Extends, Implements, Access Modifiers, public (default), private, protected, readonly, Parameter properties, abstract, static, getters/setters (like C#)

*Example:*

abstract class Animal {  
 static ***animalCount***: number = 0;  
 **protected legs**: number = 0;  
 constructor(*private* **name**: string) {  
 **Animal**.**animalCount**++;  
}  
move(met) {  
 ***console***.log(${this.name} moved ${met} m with ${this.legs});  
}  
}  
class Snake extends Animal {  
 move() { super.move(5);}  
}  
class Horse extends Animal {  
 constructor(*name*: string, *legs*: number) {  
 super(*name*);  
 this.**legs** = *legs*;  
 }  
 move() { super.move(45); }  
}  
let sam = new Snake(**"Sammy the Python"**);  
let tom: Animal = new Horse(**"King"**, 4);  
sam.move();  
tom.move(34);  
***console***.log(Animal.***animalCount***); //2