Explain the two strategies for improving JavaScript: ES6 (es2005) + ES7, versus Typescript. What does it require to use these technologies: In our backend with Node, in (many different) Browsers.

Fælles for både ES6 og typescript er at de ikke I deres *ummidbare* forstand understøttes af browseren (I hvert fald ikke game modeller). Det gør at vi I begge tilfælde bliver nødt til at få vores ES6 eller typescript kode lavet om til ES5 vanilla Javascript. Hos typescript bruger man *typeScript compileren* (kaldet tsc), der kan compile typescript filer til pure javascript. Her fra tsconfig.json

```
"compilerOptions": {
 "module": "commonjs",
   "target": "es5",
"noImplicitAny": false,
  "sourceMap": false}
```

Som man måske kan gætte sig til, så indikerer *target* fæltet hvilken version af JS man ønsker komplileret til. ES5 er et godt valg da alle browsere understøtter det (realistisk set).

ES6 bruger en lignende strategi, dog med en transpiler (tsc er også arguably en transpiler – som igen blot er en subtype af compilers). Ligesom med typescript skriver den ES6 koden (som ikke er understøttet) om til ES5.

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

LET:

```
function allyIlliterate() {
    //tuce is *not* visible out here

    for( let tuce = 0; tuce < 5; tuce++ ) {
        //tuce is only visible in here (and in the for() parentheses)
    }

    //tuce is *not* visible out here
}

function byE40() {
    //nish *is* visible out here

    for( var nish = 0; nish < 5; nish++ ) {
        //nish is visible to the whole function
    }

    //nish *is* visible out here
}
```

ARROW FUNCTIONS:

```
Promise.all([p1, p2, p3, p4, p5, p6]).then(arr =>
{
     result.randoms = arr;
     console.log(result)
});
```

er ækvivalent til at skrive:

```
Promise.all([p1, p2, p3, p4, p5, p6]).then(function(arr)
{
     result.randoms = arr;
     console.log(result)
});
```

THIS:

Når man bruger arrow functions beholder this sit scope (altså uden for arrow functionen).

REST PARAMETERS:
Er bare varArgs som vi kender dem fra Java (f.eks fra main der altid tager en varArg af String argumenter)

Her er et eks på en funktion der bruger to alm args og en varArg, og derefter printer en sum, samt alle varargs og deres plads I arrayen:

```
var john = function(x, y, ...rest)
{
     console.log(`the sum of x and y is ${x+y}`);
     rest.map((v, i) =>
     {
        console.log(`value ${i} er af typen: ${typeof v}` )
     });
}
```

MAPS:
se ovenstående eksempel for brugen af map.

Explain and demonstrate how es2015 supports modules (import and export) similar to what is offered by NodeJS.
Her er et eks, med funktionen john fra ovenstående opgave (den med rest parameters)
Exporter.js

```
module.exports ={
     f : john
}
```

importer.js

```
let f = require("./ex5");
f.f(2,5, true,2,"hello World",[1,2,3],new Date(),{});
```

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

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

Eks:

```
class Shape {
     constructor(color)
     {
          this._color = color;
     }
     getArea()
     {
          return undefined;
     }
     getPerimeter()
     {
          return undefined;
     }
     toString()
     {
          return `this shape is of the color ${this._color}, it has an undefined area and
perimeter!!`;
     }
     getColor()
     {
          return this._color;
     }
     setColor(color)
     {
          this._color = color;
     }
}

class Circle extends Shape
{
     constructor(radius, color)
     {
          super(color);
          this._radius = radius;
     }
     getArea()
     {
          return undefined;
     }
     getPerimeter()
     {
          return undefined;
     }
     toString()
     {
          return `this circle is of the color ${this._color}, a radius of ${this._radius}. It
has an undefined area and perimeter!!`;
     }
     getRadius()
     {
          return this._radius;
     }
     setRaduis(radius)
     {
          this._radius = radius;
```

```
    }
}
```

Ved brug af class er det rimelig meget ligesom Java, du kan dog også arve straight fra et objekt Istedet for en klasse, hvilket er relativt unjavalike.

Explain about Generators and how to use them to:
• Implement iterables
• Implement blocking with asynchronous calls

Iterables kan bedst forklares med følgende:

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
}
var it = foo();
var itterator = it.next(); //første run giver 1 next giver 2 osv.
```

Du kan bruge den med async kald, ved at kalde next I then funktionen af et async promise (og yielde det næste promise), således at det først bliver kørt når det første er done.

Explain about promises in ES 6, Typescript, AngularJS including:
• The problems they solve:

Promises løser et essentielt problem som "non-blocking" sprog/apis har nemlig, hvad gør man hvis man har en operation der SKAL eksekveres på et specifikt tidspunkt. Non-blocking adfærd er ok, når man bare vil lave alerts på en hjemmeside, skal man dog lave evt et http-request, hvor resultatet bruges I et databasekald (begge async operationer), bliver det lige pludseligt vigtigt, at du kan lave http-requestet først (hver gang). Promises kan da bruges på følgende måde (I forhold til ovenstående eksempel): "*Lav et http-request og **lov** mig, at du vil gemme resultatet I følgende variabel, når du er færdig – der så kan bruges I DB kaldet"* (er det underligt at snakke sådan til sin computer?). I korte træk tillader promises altså udvikleren, at lave asynkone kald I et singletrådet, non-blocking miljø.

• **Examples to demonstrate how to avoid the "Pyramid of Doom"**
Eks: vi ønsker en række tal fra en async operation, I en specifik rækkefølge, for at sikre det skal man normalt gøre følgende (psudokode):

```
getNumber(funktion(array, callback)
{
var asyncCall
callback(asyncCall)
}
```

Hvis vi ville have say 6 tal, skulle vi da kalde denne funktion seksgange inde I den første, således at vi får nedestående eller lign:

```
func(){
    func(){
        func(){
            func(){
                func(){
                    }
                }
            }
        }
    }
```

I stedet kan man bruge pormises og gøre følgende (eks fra opgave), derfor med en smart lille promiseFactory:

```
let makeRandoms = function (callback)
{
    let p1 = promiseFactory(48);
    let p2 = promiseFactory(40);
    let p3 = promiseFactory(32);
    let p4 = promiseFactory(24);
    let p5 = promiseFactory(16);
    let p6 = promiseFactory(8);
    let result =
    {
        title: "6 secure randoms",
        randoms: []
    }
    Promise.all([p1, p2, p3, p4, p5, p6]).then(arr =>
    {
        result.randoms = arr;
        callback(result);
    });
}
```

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.
Efter V8 (google JS runtime) er det perfomance-mæssigt viable at skrive seriøse applikationer I JS. Dog er JS designet af en mongol med et voldsomt alkoholproblem. Derfor er der sprunget mange frameworks op, for at tilføre JS en grad af fornuft, således at udviklerene forhåbentligt, kan arbejde et par år mere, før de bliver sindsyge. Typescript er et af disse frameworks (ligesom jQuery, Angular, CoffeeScript osv). Den allerstørste forskel på vanilla JS og typescript er (som navnet indikerer) typesikkerhed. Hvor JS opererer med en type var, har typescript faktiske datatyper, og en transpiler der giver en warning, hvis du assigner f.eks en boolean til et number field.

Da TS compiles til JS (ES5), kan du ez køre det på alle clienter – same for server-side, da du altid bare kan lave det til JS og smide det på en express server.

Provide examples of Interfaces with typescript and explain what is meant by the term duck-typing.
Først duck-typing. Begrebet duck-typing bruges om ej typefaste sprog og stammer fra det engelske udtryk: "If it looks like a duck and quacks like a duck, it's a duck". I korte træk vil det sige, at du kan gøre følgende:

randomMetode(helTal){return (helTal/75)*27}

randomMetode(“abekat”);

Ovenstående er tydeligvis åndsvagt (hvordan dividerer man “abekat” med 75?), dog får udvikeren ikke nogen error før runtime. Dette står I modsætning til strongly typed sprog, hvor du ikke ville få lov at compile koden, hvis du satte en String ind. Typescript interfaces giver dig mulighed for at definerer en række KRAV til et objekt, således at du kan gøre følgende:

```
interface Food {
    name: string;
    calories: number;
}

function speak(food: Food): void{
    console.log("Our " + food.name + " has " + food.calories + " calories.");
}

var ice_cream = {
    name: "ice cream",
    calories: 200
}

var ice_creamError = {
    nmea: "ice cream",
    calories: 200
}
```

Her ville compileren ikke give dig lov til at kalde speak med objektet ice_creamError, da speak tager en Food, og ice_creamError ikke overholder “reglerne” for Food.

Explain TypeScript Generics, the problems they solve, and provide examples of your own generic functions and classes.

Generiske datatyper lader dig give en funktion en hvilken som helst data som input.