Forskellen på Java og Javascript:

Et program der er lavet i Java er nødt til at compile før det kan køre, mens et Javascript program ikke behøver at compile først. Dette er fordi Javascript er et scripted program.

Java laver applikationer som kører i en virtuel maskine eller browser, mens Javascript kode kun bliver kørt i en browser.

Opgaver:

1. Strict mode er for til at indikere at koden skal blive eksekveret i “strict mode”. I ”strict mode” kan man for eksempel ikke bruge udeklarerede variabler.  
     
   x = 3.14;       // This will not cause an error.   
   myFunction();  
     
   function myFunction() {  
      "use strict";  
       y = 3.14;   // This will cause an error (y is not defined)  
   }
2. I Javascript er der “hoisting”. Dette vil sige at funktioner bliver ”komplet hoisted”. Altså kan man køre en funktion selvom den først bliver deklareret længere nede.   
   Det samme gør sig gældende ved variabler dog bliver de ikke komplet hoisted. Det vil sige man kan godt ”bruge” en variabel før den er deklareret, men hvis den vil være ”undefined”.

Se feks. Opgave 5 på github linket.

1. This i java er en reference til det nuværende object der er blevet instansieret af classen.   
   This i Javascript referer i en global kontekst (altså når this ikke er i en funktion) til det globale object.   
   Når this er inde i en funktion så kommer det an på hvordan funktionen bliver kaldt.  
     
   console.log(this.document === document); // dette er true.
2. Closure er hvor man har kontrol over hvorvidt en variabel er tilgængelig globalt eller lokalt.   
   Det vil sige en variabel deklareret udenfor en funktion er globalt tilgængelig, mens en variabel deklareret indeni en funktion kun er tilgængelig i den funktion (altså lokalt scope for den funktion).  
     
   variable1 = 1; // Global Scope

var variable2 = 2; // Not within a function: Global Scope

function funcName() {

variable3 = 3; // No var keyword: Global Scope

var variable4 = 4; // Local Scope only

console.log(variable1);

}

funcName();

console.log(variable3); // returnerer 3

console.log(variable4); // returnerer ReferenceError: variable4 is not defined.

1. Det er lidt ligesom at lave getters og setters private. Man kan kalde og bruge ”modularpattern” men man har ikke adgang til variablerne.  
     
   var modularpattern = (function() {

// your module code goes here

var sum = 0 ;

return {

add:function() {

sum = sum + 1;

return sum;

},

reset:function() {

return sum = 0;

}

}

}());

alert(modularpattern.add()); // alerts: 1

alert(modularpattern.add()); // alerts: 2

alert(modularpattern.reset()); // alerts: 0

1. Når man bruger Immediately-Invoked Function Expression så betyder det at funktionen bliver kørt med det samme. Putter man den funktion i en variabel så betyder det variablen indenholder resultated og ikke funktionen.  
     
   var a = (function(){  
    var b = 10;  
    return b;

}());  
// A er her = 10.

1. Alle Javascript objekter har en prototype. Prototypen er også et objekt. Alle JS objekter arver deres properties og metoder fra deres prototype.   
     
   Javascripts prototype er deres skelet (læs constructor). Et skelet man kan tilføje til, både properties og methods. Feks:  
     
   function Person(first, last, age, eyecolor) {  
       this.firstName = first;  
       this.lastName = last;  
       this.age = age;  
       this.eyeColor = eyecolor;  
   }  
   Person.prototype.name = function() {  
       return this.firstName + " " + this.lastName;  
   };
2. En callback function er en function som bliver ”passed” til en anden function som et parameter. Den callback function er så kaldt (kørt) inden i den anden function.  
     
   var names = ["kurt","ole","ib"];

function myforEach(arr,callback){

for(var i =0; i < arr.length; i++){

callback(arr[i]);

}

}

myforEach(names,function(name){

console.log(name);

});

1. **npm** is a NodeJS package manager. As its name would imply, you can use it to install node programs. Also, if you use it in development, it makes it easier to specify and link dependencies.  
     
   **Node**.**js** is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. **Node**.**js** uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.
2. **Event Loop**

The Node.js event loop runs under a single thread, this means the application code you write is evaluated on a single thread. Nodejs itself uses many threads underneath through libuv, but you never have to deal with with those when writing nodejs code.

Every call that involves I/O call requires you to register a callback. This call also returns immediately, this allows you to do multiple IO operations in parallel without using threads in your application code. As soon as an I/O operation is completed it's callback will be pushed on the event loop. It will be executed as soon as all the other callbacks that where pushed on the event loop before it are executed.

1. Man laver sin egen funktion som et node.js modul. Dette kan man så ”require” og bruge i sine node.js filer.