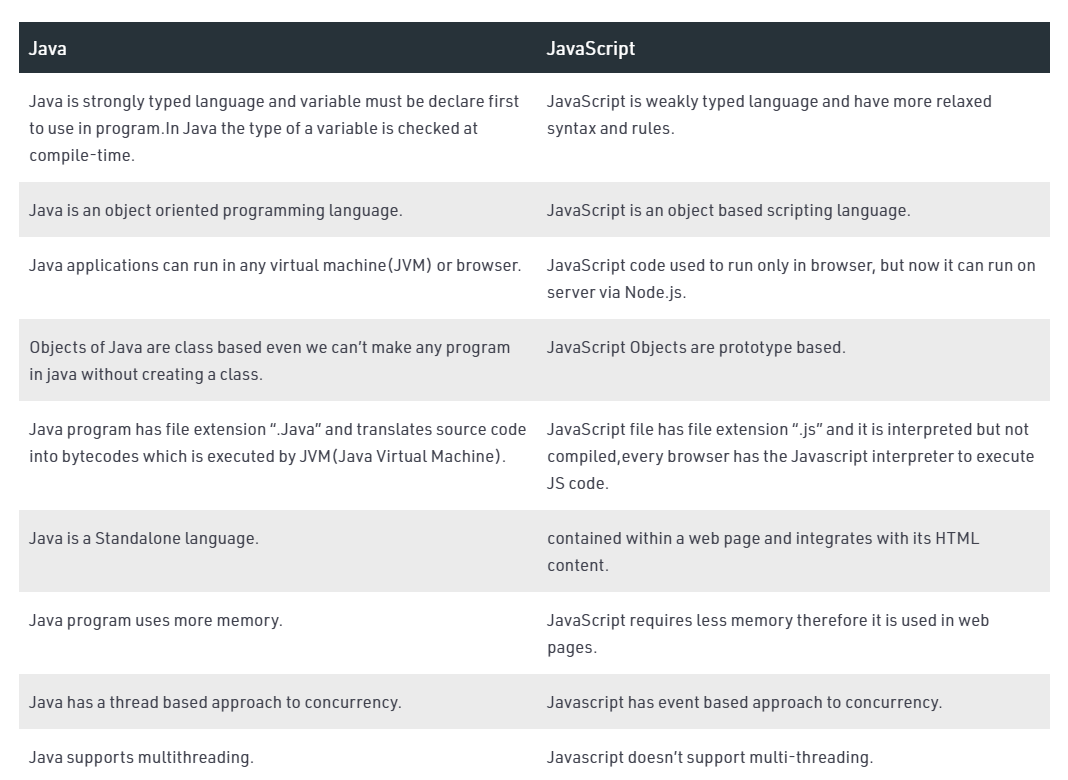
**Period-1 Vanilla JavaScript, Es-next, Node.js, Babel + Webpack and TypeScript-1**

Note: This description is too big for a single exam-question. It will be divided up into several smaller questions for the exam

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**  
    Blocking henviser til operationer, der blokerer for yderligere udførelse, indtil den handling er afsluttet, mens ikke-blokering henviser til kode, der ikke blokerer udførelse. Eller som Node.js docs udtrykker det, er blokering, når udførelsen af yderligere JavaScript i Node.js-processen skal vente, indtil en ikke-JavaScript-handling er afsluttet.

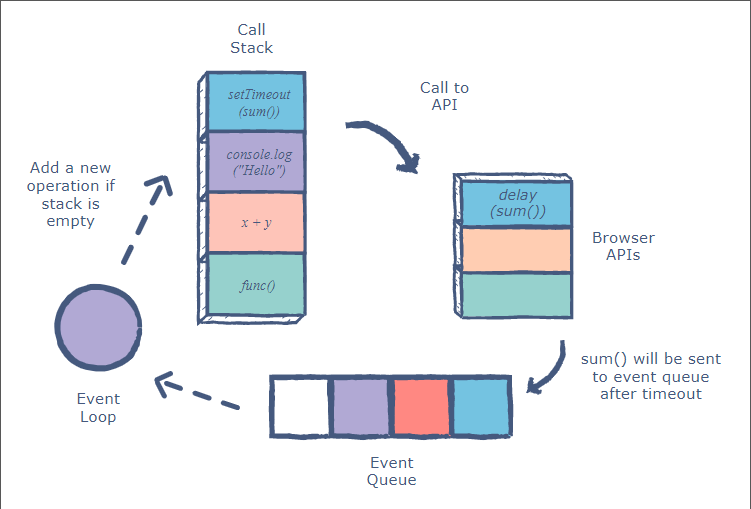
Blocking udføres synkront, mens non-blocking metoder udføres asynkront.

* **Explain generally about node.js, when it “makes sense” and *npm*, and how it “fits” into the node echo system.**  
  **Node**

Du vil bestemt ikke bruge Node.js til CPU-intensive operationer; faktisk, ved at bruge det til tunge beregninger vil annullere næsten alle dets fordele. Hvor Node virkelig skinner, er at opbygge hurtige, skalerbare netværksapplikationer, da det er i stand til at håndtere et stort antal samtidige forbindelser med høj kapacitet, hvilket svarer til høj skalerbarhed.

Sammenlignet med traditionelle webeknikker, hvor hver forbindelse (request) starter en ny tråd, optager system-RAM og til sidst bruger alt den tilgængelige mængde RAM, fungerer Node.js på en enkelt tråd ved hjælp af non-blocking I / O-opkald, der gør det muligt at understøtte titusinder af samtidige forbindelser, der holdes i hændelsesløkken.   
Derfor supergodt sammen med ex. Express (webserver)  
  
**NPM**

NPM står for Node Package Manager. Det er hvad navnet beskriver. Det er pakkehåndtering til node-baserede miljøer. Det holder styr på alle pakkerne og deres versioner og giver udvikleren mulighed for nemt at opdatere eller fjerne disse afhængigheder. Alle disse eksterne afhængigheder opbevares i en fil, der hedder package.json.

* **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.**  
    
  **Call Stack** er ansvarlig for at holde styr på alle operationer i kø, der skal udføres. Hver gang en funktion er færdig, poppes den fra stakken.  
  **Callback Queue / Event Queue** er ansvarlig for at sende nye funktioner til processoren til behandling. Det følger kødatastrukturen for at opretholde den korrekte rækkefølge, hvor alle operationer skal sendes til udførelse.
* **What does it mean if a method in nodes API's ends with xxxxxxSync?**  
  At den er “Blocking”
* **Explain the terms JavaScript Engine (name at least one) and JavaScript Runtime Environment (name at least two)**  
  En **Javascript Engine** er et program eller en tolk(interpreter), der udfører JavaScript-kode. En JavaScript-motor kan implementeres som en standardtolk eller just-in-time-kompilator, der kompilerer JavaScript til bytekode i en eller anden form.  
  Den mest brugte er **V8**, udviklet af Google.  
    
  **Javascript runtime** environment giver forskellige funktioner / API'er til at opbygge Javascript-baseret software.  
  **Browser**: Giver DOM API, Fetch API, Timer (setTimeout & setInterval), Storage (som Local Storage) osv.

Eksempel: Chrome, Firefox, Safari, Opera, Edge osv  
  
**Servermiljø**: Giver filsystemadgang, netværksadgang, konsol osv.

Eksempel: NodeJS, Deno  
  
**Desktopmiljø**: Tilbyder GUI API, File System Access, Network Access, Console osv.

Eksempel: Elektron osv.  
  
**Mobilt miljø**:

Eksempel: React Native, NativeScript, Ionic, PhoneGap osv.

* **Explain (some) of the purposes with the tools *Babel* and *WebPack and how they differ from each other*.       Use examples from the exercises.  
  Babel** er et værktøj, der hovedsagelig bruges til at konvertere ECMAScript 2015+ kode til en bagudkompatibel version af JavaScript i nuværende og ældre browsere eller miljøer.  
    
  Som kerne er **webpack** en statisk modulbundler. I et bestemt projekt behandler webpack alle filer og aktiver som moduler. Under emhætten er den afhængig af en afhængighedsgraf. En afhængighedsgraf beskriver, hvordan moduler relaterer til hinanden ved hjælp af referencer (require / import) mellem filer. På denne måde krydser webpack statisk alle moduler for at opbygge grafen og bruger den til at generere et enkelt bundt (eller flere bundter) - en JavaScript-fil, der indeholder koden fra alle moduler kombineret i den rigtige rækkefølge. "Statisk" betyder, at når webpack bygger sin afhængighedsgraf, udfører den ikke kildekoden, men syr moduler og deres afhængigheder sammen i en pakke. Dette kan derefter inkluderes i dine HTML-filer.

**Explain using sufficient code examples the following features in JavaScript (and node)**

* **Variable/function-Hoisting**  
  DAY\_1 function-Hoisting.js
* ***this* in JavaScript and how it differs from what we know from Java/.net.**DAY\_1 this.js
* **Function Closures and the JavaScript Module Pattern  
  Function Closures:**DAY\_1 function-closures.js  
    
  **Module pattern:**  
  It is used to define objects and specify the variables and the functions that can be accessed from outside the scope of the function. We expose certain properties and function as public and can also restrict the scope of properties and functions within the object itself, making them private.  
  DAY\_1 modulepattern.js
* **User-defined Callback Functions (writing functions that take a callback)**  
  DAY\_1 day1.jsex 2 a
* **Explain the methods map, filter and reduce**  
  **Map, filter** DAY\_1 ex b / c  
  **Reduce** DAY\_1 the reduce-method
* **Provide examples of user-defined reusable modules implemented in Node.js**
* **Provide examples and explain the es2015 features: let, arrow functions, this, rest parameters, destructuring objects and arrays,   maps/sets etc.**- Examples in DAY\_3 index.js  
  The **let** statement declares a block-scoped local variable, optionally initializing it to a value.  
    
  **Arrow functions** are meant for short pieces of code that do not have their own “context”, but rather work in the current one. And they really shine in that use case.

With arrow functions the **this** keyword always represents the object that defined the **arrow function**.  
  
The **rest parameter** syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent.  
A function definition can have only one ...restParam.  
The rest parameter must be the last parameter in the function definition.  
  
The **destructuring** assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

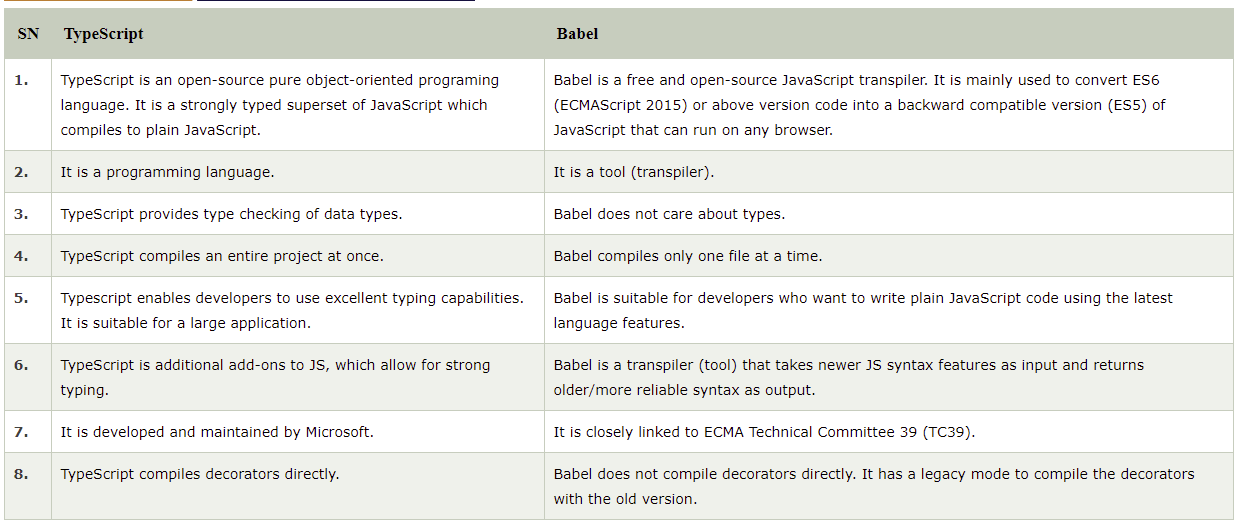
* **Provide an example of ES6 inheritance and reflect over the differences between Inheritance in Java and in ES6.**

MISSING

* **Explain and demonstrate, how to implement event-based code, how to emit events and how to listen for such events**- Examples in DAY\_3 index.jsevent-driven programmering som bruges oftest i front-end udvikling, hvor den ene begivenhed kan formeres gennem systemet og mange aktører kan handle ud fra at fange denne begivenhed. Datastrømmen er envejs, og tilføjelse af nye funktioner kan udføres uden at redigere eksisterende komponenter.

Mens event-driven programmering er dominerende for opbygningen brugergrænseflader, kan vi bruge det til at skrive server-side kode også. Tilfælde med god brug er meget asynkrone systemer, der ikke kræver et øjeblikkeligt svar fra serveren og bruger forskellige kommunikationskanaler til at offentliggøre forløbet af en anmodning.

**ES6,7,8,ES-next and TypeScript**

* **Provide examples with es-next, running in a browser, using Babel and Webpack  
  - MISSING**
* 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  
    
  Both require installation and some configuration
* **Provide examples to demonstrate the benefits of using TypeScript, including, types, interfaces, classes and generics**  
  DAY\_5 typescript.js

* **Explain how we can get typescript code completion for external imports.**Try to install them with  
  <https://www.npmjs.com/package/@types/packageName>  
    
  Else create your own declaration for it.

## Explain the ECMAScript Proposal Process for how new features are added to the language (the TC39 Process) 🤷🏻 Stage 0: Strawperson

This is the first [stage](https://github.com/tc39/proposals/blob/master/stage-0-proposals.md), called “Strawperson”, representing an **initial idea** for addition or change to the specification that isn’t considered as a formal proposal. Suggestions for this stage must come from a TC39 member or registered contributor.

💡 Stage 1: Proposal

This [stage](https://github.com/tc39/proposals/blob/master/stage-1-proposals.md) is a **formal proposal** that describes a discrete problem or general need, suggests a shape of the solution and points out potential challenges - such as “cross-cutting” concerns with other features or complex implementation. The solution’s description should contain a high-level API with concrete examples; and also discuss algorithms, abstractions and semantics.

On top of that, one of the TC39 members is defined as the owner that is responsible to advance the proposal, and practically named champion. Typically the champion is the original author of the proposal, but not always. If the proposal meets the criteria of stage 1, and hereby representing the committee’s will of going forward with the proposal, then it moves to the draft stage.

✍🏻 Stage 2: Draft

This [stage](https://github.com/tc39/proposals#stage-2) is the **initial draft** of the proposal in the specification, phrased by the ECMAScript language.

The draft should describe the syntax, semantics and APIs precisely as much as possible, although it can have “TODO” comments or placeholders. An experimental implementation is also needed, runnable by a browser or a build-time transpiler like Babel.

Moving forward from this stage means that the committee expects that the proposal would be developed and included eventually in the official specification - when only incremental changes (and mostly fixes) are expected.

📝 Stage 3: Candidate

This [stage](https://github.com/tc39/proposals#stage-3) is a candidate proposal that’s **almost final** - but ready for feedback and refinements from implementations and users. The proposal defined as completely final when there is neither further work with the specification nor external feedback.

All ECMAScript editors and designated reviewers should sign off on this specification. In addition, it should include two independent spec-compatible implementations passing the [acceptance tests](https://github.com/tc39/test262).

After this stage, changes would be made only for critical issues.

✅ Stage 4: Finished

This is the last [stage](https://github.com/tc39/proposals/blob/master/finished-proposals.md), called “finished” obviously, indicating that the proposal is **ready** to be included in the latest draft of the specification - and be **delivered** with its next edition.

**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:**A **Promise** is 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.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

* ~~Example(s) that demonstrate how to avoid the callback hell  (“Pyramid of Doom")~~Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively.  
  Example DAY\_3 callback-hell.js
* Example(s) that demonstrate how to implement **our own** promise-solutions.  
  DAY\_3 promises.js
* Example(s) that demonstrate error handling with promises

DAY\_3 promise-errors.js

* Example(s) that demonstrate how to execute asynchronous (promise-based) code in **serial** or **parallel**DAY\_3 serial-parallel.js

**Explain about JavaScripts async/await, how it relates to promises and reasons to use it compared to the plain promise API.**  
Syntactic sugar on top of promises, making asynchronous code easier to write and to read afterwards. They make async code look more like old-school synchronous code, so they're well worth learning.  
- easier to understand what's going on if the code simply looks as though it's synchronous

Provide examples to demonstrate

* **Why this often is the preferred way of handling promises**
* **Error handling with async/await**  
  For both of above see   
  DAY\_3 promise-errors.js
* Serial or parallel execution with async/await.

DAY\_3 serial-parallel.js

Se the exercises for Period-1 to get inspiration for relevant code examples