**Laborationsrapport**

**Moment 4 / ECMA-script och Typescript**

*DT173G, Webbutveckling III*

**Författare: Jenny Sundqvist,** [**jens2001@student.miun.se**](mailto:jens2001@student.miun.se) **Termin, år: HT, 2021**



# Sammanfattning

Syftet med denna inlämningsuppgift är att beskriva funktionalitet i Ecmascript samt övriga varianter av ramverk för att generera JavaScript, samt att förstå vad TypeScript är och hur det skiljer sig från JavaScript.

# Innehållsförteckning

[1 Sammanfattning 2](#__RefHeading___Toc213_2266673292)

[2 Innehållsförteckning 3](#__RefHeading___Toc217_2266673292)

[3 Frågor 4](#__RefHeading___Toc219_2266673292)

[3.1 Förklara begreppet "Ecmascript", samt dess historia för JavaScript. Gå igenom vilka olika versioner som kommit genom åren, och nämna några av de större uppdateringarna som kommit med vardera version. 4](#__RefHeading___Toc221_2266673292)

[3.2 Förklara följande tekniker, med kod-exempel för vardera: 4](#__RefHeading___Toc229_2266673292)

[3.3 Förklara Fetch API och Promises 7](#__RefHeading___Toc243_2266673292)

[3.4 För bakåt-kompabilitet - hur kan vi konvertera senare versioner av Ecmascript till något som fungerar i "alla" webbläsare med Gulp (alternativt Webpack)? Uppdatera ditt Gulp- eller Webpack-projekt från tidigare laboration(er) och publicera denna till Github eller annan webbtjänst för Git. Lägg in kommandot för att klona ditt repo till lokal dator som en del av svaret på denna fråga. 8](#__RefHeading___Toc245_2266673292)

[3.5 Gör en analys av Typescript. Analysen bör vara i storleksordningen en till två A4-sidor. Använd kod-exempel och illustrationer.  Analysen ska innehålla historia samt användningsområde(n). Vad gör det användbart? Vad kan det användas till? Ge kod-exempel på saker du tycker är viktiga/användbara, där Typescript har fördelar jämtemot JavaScript (skapa egna exempel, använd inte färdiga från respektives webbplats).  Beskriv även hur utvecklings-miljön fungerar för Typescript. 9](#__RefHeading___Toc247_2266673292)

[4 Slutsatser 12](#__RefHeading___Toc249_2266673292)

[5 Källförteckning 13](#__RefHeading___Toc251_2266673292)

# Frågor

## 3.1 Förklara begreppet "Ecmascript", samt dess historia för JavaScript. Gå igenom vilka olika versioner som kommit genom åren, och nämn några av de större uppdateringar som har kommit med vardera version.

”Ecma” som står för ”European Computer Manufacturers Association” är en organisation för standardisering av informations- och kommunikationssystem. Ecmascript är standardiserat av Ecma och anger standarden för skriptspråk. Syftet med standardisering är att se till att den kod man skapar ska fungera i alla webbläsare. JavaScript och Ecmascript är i grunden samma språk. Ecmascript är inte bara för webbläsare utan även för NodeJs (ex npm, gulp och serverbaserad programmering).

EcmaScript har utvecklats mycket sedan starten och har hittills kommit i elva olika versioner. Den första versionen av EcmaScript kom i juni 1997 efter att Netscape lämnat in JavaScript till Ecma för standardisering av språket så att det skulle bli kompatibelt med alla webbläsare. Den nuvarande versionen kom i juni 2021, det är version 12 - ECMAScript 202. Version 4 ratades (2003) innan den blev klar, på grund av politiska meningsskiljaktigheter gällande språkets komplexitet. Varje version innehåller förbättringar och nya funktioner i språket som ska underlätta för användaren.   
**Version 2** (1998), här gjordes mest redaktionella ändringar för att hålla internationell standard enligt ISO/IEC 16262  
**Version 3** (1999), tillägg av regular expressions, bättre stränghantering, nya control statements, formatering för numerisk output m.m.  
**Version 5** (2009), här lade man bl.a. till möjligheten att köra i strict mode, getters och setters och support för JSON.   
**Version 6** (2019), man lade bl.a. till let och konst samt olika array-funktioner  
**Version 7** (2016), här lade man exempelvis till exponential operator \*\* för nummer samt async-nyckelord för asynkron programmering (som förberedelse för nästa version)  
**Version 8** (2017), här lades bl.a. async-funktioner till samt string padding och funktioner för objekthantering  
**Version 9** (2018), nya funktioner inkluderade t.ex. spread operator (underlättar kopiering av objektegenskaper), rest-parametrar och asynkron iteration (upprepning)  
**Version 10** (2019), här gjordes främst flera ändringar och tillägg för array-funktioner  
**Version 11** (2020), tillägg av bl.a. BigInt, ett inbyggt objekt för att hantera numeriska funktioner samt nullish coalescing operator (??)  
**Version 12** (2021), den aktuella versionen som bl.a. lade till replaceAll-metod för strängar och AggregateError som kan representera flera fel samtidigt  
[1][2]

## 3.2 Förklara följande tekniker, med kod-exempel för vardera:

**Classes**  
En klass är en mall för att skapa objekt. Den kapslar in data med kod för att arbeta med datat. Klasser är specialfunktioner, klassyntaxen har två komponenter, *class expressions* och *class declarations*.

En *class expression* kan vara namngiven:  
let Rectangle = class Rectangle2 {

constructor(height, width) {

this.height = height;

this.width = width;

}

};  
eller icke-namngiven:  
let Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

En *class declaration* är ett annat sätt att definiera en klass:

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}[3]

**Higher order functions**

Higher-order functions är funktioner som använder andra funktioner, antingen genom att dessa används som argument eller genom att de returneras. Det är en funktion som antingen accepterar en funktion som en parameter eller returnerar funktionen som en output.   
Exempel på funktion som skapar ny funktion:  
function greaterThan(n) {

return m => m > n;

}

let greaterThan10 = greaterThan(10);

console.log(greaterThan10(11));

// → true  
  
[4]

**Higher order array methods - forEach, map och filter**

**forEach** är en inbyggd array-metod som skapar en slags for/of-loop som en higher-order function. Den loopar genom varje element i arrayen, ett i taget, och utför en åtgärd. Om man använder en forEach-loop så ändras datat på originalarrayen.

var number = [1, 2, 6, 8, 12, 14];  
  
number.forEach((element, index) => {  
console.log('index: ' + index + ' number: ' + element)  
})  
/\* Expected Output:

index: 1 number: 2

index: 2 number: 6

index: 3 number: 8

index: 4 number: 12

index: 5 number: 14 \*/

**Map** är en metod som tar en callback-funktion och kör den för varje element i arrayen. Den här funktionen ändrar inte originalarrayen utan skapar ytterligare en array med nya värden.

var number = [1, 2, 6, 8, 12, 14];

number.map(element => element\*element)  
console.log(square);

/\* Expected Output:  
[1, 4, 36, 64, 144, 196]  
\*/  
  
**Filter** är en metod som skapar en ny array med de element som klarar filtreringen som har satts i callback-funktionen, den används alltså för att filtrera en given array och skapa en ny array med utvalda värden.

var number = [1, 2, 6, 8, 12, 14];  
  
number.map(element => element>5)  
console.log(result);  
  
/\* Expected Output:  
[6, 8, 12, 14]  
\*/

[5]

**Spread operators ("spreads")**

Spread operatorn (…) används för att expandera eller breda ut (spread) en array eller iterator.

const arrValue = ['My', 'name', 'is', 'Jack'];

console.log(...arrValue); // My name is Jack

[6]

**Destructuring**Ett uttryck inom JavaScript som gör det möjligt att packa upp värden från arrayer eller egenskaper från objekt, till specifika variabler.

// assigning object attributes to variables

const person = {

name: 'Sara',

age: 25,

gender: 'female'

}

// destructuring assignment

let { name, age, gender } = person;

console.log(name); // Sara

[7]

**Arrow Functions**

Arrow functions är en relativt ny funktion inom JS som gör att man kan skriva renare och mindre kod.   
I stället för att skriva traditionellt så här:  
let x = function(x, y) {

return x \* y;

}

Kan man skriva så här med en arrow-funktion och uppnå samma resultat:  
let x = (x, y) => x \* y;  
[8]

## 3.3 Förklara **Fetch API** och **Promises**

Ett REST API skickar eller hämtar data från en databas eller tredjeparts-API. Ett REST API är uppbyggt av tre element; request (datat som skickas till API:t), response (datat som kommer tillbaka efter en request) och headers (kompletterande metadata för att förtydliga vilken typ av request servern tar emot).

Fetch API är en senare och enklare variant av AJAX (Asynchronous JavaScript and XML) med XMLHttpRequest, där syntaxen är lite annorlunda. Fetch API används för att konsumera resources asynkront.

Första parametern som skickas med i Fetch är alltid URL:en. Fetch tar sedan ett JSON-objekt med alternativ som metod, headers, request, body m.m. Exempel på Fetch med GET:   
fetch('https://api.github.com/users/manishmshiva', {  
 method: "GET",  
 headers: {"Content-type": "application/json;charset=UTF-8"}

})  
.then(response => response.json()) //Konvertera till json  
.then(json => console.log(json)); //Skriv ut data till konsol  
.catch(err => console.log(err)); //Fånga upp fel

Fetch API använder **Promises** i stället för callbacks som vid AJAX, vilket ger en enklare och renare API. En promise är ett JavaScript-objekt som representerar den eventuella färdigställandet eller misslyckandet av en asynkron operation samt dess resulterande värde. I stället för att direkt returnera det slutgiltiga värdet så gör den asynkrona metoden så att den returnerar ett löfte om att provide värdet vid en specifik tidpunkt i framtiden. En promise har tre stadier pending (initieringsstadiet, varken lyckat eller misslyckat), fulfilled (operationen har lyckats) eller rejected (operationen har misslyckats). Parametern resolve används när man vill att ett promise ska lyckas och reject för att fånga upp ett misslyckat promise.

Exempel på ett promise:

const makeServerRequest = new Promise((resolve, reject) => {

let responseFromServer;

if (responseFromServer) {

resolve("We got the Data");

} else {

reject("Data not Found");

}

});

*responseFromServer* representerar ett svar från servern. Om det är true kommer resolve-metoden att kallas på för att genomföra promise:t och returnera strängen som sitt argument. Annars om svaret är false så kommer promiset att misslyckas och kalla på reject-metoden.

[9][10]

## 3.4 För bakåt-kompabilitet - hur kan vi konvertera senare versioner av Ecmascript till något som fungerar i "alla" webbläsare med Gulp (alternativt Webpack)? Uppdatera ditt Gulp- eller Webpack-projekt från tidigare laboration(er) och publicera denna till Github eller annan webbtjänst för Git. Lägg in kommandot för att klona ditt repo till lokal dator som en del av svaret på denna fråga.

För att konvertera Ecmascript till ett universellt språk som fungerar i alla webbläsare kan man använda sig av en JavaScript-transpilerare, exempelvis en som heter *Babel*. Tack vare den kan man använda sig av den senaste syntaxen i TypeScript med exempelvis arrow-funktioner och Babel kommer att konvertera allt till ett skript som ändå fungerar med äldre versionen av Ecmascript. [13]

## 3.5 Gör en analys av **Typescript**. Analysen bör vara i storleksordningen en till två A4-sidor. Använd kod-exempel och illustrationer.  Analysen ska innehålla historia samt användningsområde(n). Vad gör det användbart? Vad kan det användas till? Ge kod-exempel på saker du tycker är viktiga/användbara, där Typescript har fördelar jämtemot JavaScript (skapa egna exempel, använd inte färdiga från respektives webbplats).  Beskriv även hur utvecklings-miljön fungerar för Typescript.

TypeScript är ett programmeringsspråk som utvecklades av Microsoft 2012. Anders Hejlsberg var en av frontfigurerna bakom utvecklingen av språket som är ett strikt superset av ECMAScript 2015. TypeScript har alltså alla funktioner som JavaScript men med adderad funktionalitet.

TypeScript har utvecklats för att kunna hantera utveckling av större program och innehåller mer utvecklad syntax för att stödja en tätare integration med editorn. TS ger helt enkelt en bättre utvecklarupplevelse genom att koden blir tydligare och lättare att felsöka, TS indikerar för fel i koden på ett sätt som JS inte gör. TypeScript kan kompileras till JavaScript och kan således köras överallt där JS kan köras; i en webbläsare eller Node.js osv.

I TypeScript kan man använda sig av Type Annotations, som indikerar vilken datatyp ett värde har och ger ett felmeddelande om angivna värden inte matchar mot den angivna datatypen. När man sedan kompilerar TS till JS, så tar kompilatorn bort information, som ex. type annotations, som inte JS kan hantera, annars skulle det generera fel i JS-koden.   
Exempel på type annotation:   
let CatName: string = ”Snufsen”;  
Det är dock inte nödvändigt i de flesta fall att använda sig av detta då TS är så intuitivt att det förstår vilken typ av data det handlar om, är det en sträng eller är det ett nummer osv. TypeScript har statisk typning, det vill säga en variabel kommer att tilldelas en datatyp vid kompilering och behålla den under resten av programmets livslängd. Detta gör det möjligt att tidigt upptäcka fel, om man försöker tilldela en variabel fel datatyp så kommer man att få felmeddelande. TypeScript har datatyper som *string, number, boolean, null, array, enum, tuple* och *generics.*

TypeScript liksom JavaScript stöder användningen av klasser och objekt-orienterad programmering.

I TS kan man också använda sig av *interfaces* (gränssnitt) för att definiera vilken syntax en typ eller en klass ska följa.  
Exempel på ett interface som definierar variabeltyp:  
interface Cat {  
 name: string;  
 age: number;  
 breed: string;  
}  
let Snufsen: Cat = {name: ”Snufsen”, age: 16, breed: ”British shorthair”}; // Kompilerar OK  
let Bosse: Cat = {name: ”Bosse”, age: ”Tolv”, breed: ”Bondkatt”}; //Ger kompileringsfel då age ska vara av typen nummer, inte sträng.

Exempel på interface som definierar klass  
interface Flowers {  
 type: string;  
 price: number;

}  
class Bouquet {  
 type: string;  
 price: number;

constructor (type: string, price: number){  
 this.type = type;  
 this.price = price;

}

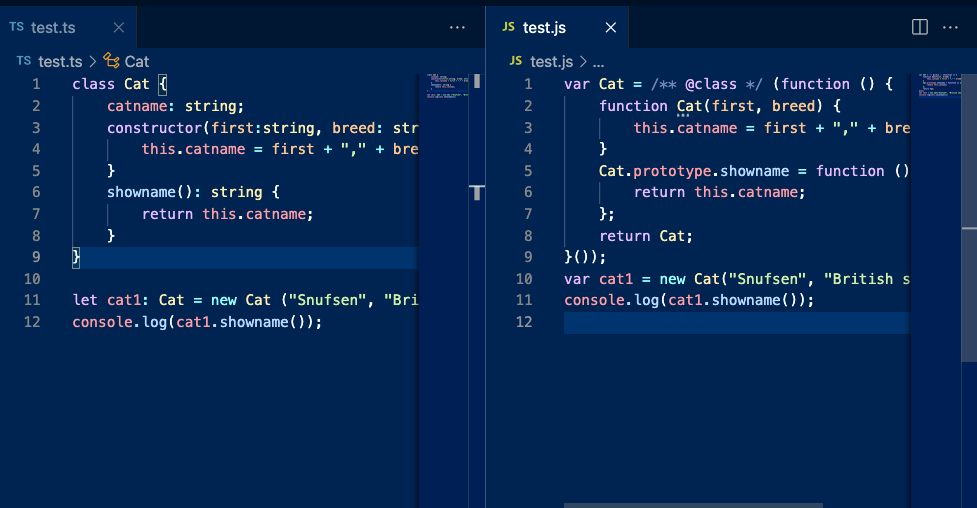
}  
let flower: Flower = new Bouquet (”Red rose”, 50);

Funktionen *generics* finns i TS men inte i JS. Generics tillhandahåller variabler till typer. Används vanligtvis för att definiera en array för att beskriva de värden som den innehåller.  
Exempel:  
type CatNameArray = Array<string>;

*Enums* är också unika för TypeScript jämfört med JavaScript, däremot finns de också i exempelvis Java och C. Enums möjliggör för en utvecklare att definiera ett set av namngivna konstanter och de kan vara både numeriska och i sträng-form.   
Exempel på en numerisk enum:  
enum Latitude {  
 East,   
 West,  
 South,  
 North,  
}  
Här kommer varje värde automatiskt att tilldelas ett nummer genom auto-increment, East = 0, West = 1 osv.

TS använder sig av en konfigurationsfil, tsconfig.json, som innehåller information om projektet och vad kompileraren förväntas göra, exempelvis var de genererade js-filerna ska skapas och vilken JS-version som stöds. När man kör kommandot tsc så letar kompileraren efter denna fil. Vid kompileringen kommer TS att varna för ev problem, exempelvis om man använder sig av funktioner som inte stöds av angiven JS-version.

Här är ett exempel på TypeScript-kod som har kompilerats till JavaScript-kod:



[11][12]

# Slutsatser

Vilka slutsatser har du dragit av uppgiften?

# Källförteckning

Här följer exempel på hur en källförteckning kan utformas enligt Vancouver-systemet. Den är automatiserad enligt metoden numrerad lista och korsreferenser. Radera denna text, samt ersätt källorna med dina egna.

1. ECMA-script, Wikipedia,  
   [https://en.wikipedia.org/wiki/ECMAScript#ES2016](https://en.wikipedia.org/wiki/ECMAScript" \l "ES2016)  
   Hämtad 2021-10-15
2. JavaScript Versions, W3Schools,  
   <https://www.w3schools.com/js/js_versions.asp>  
   Hämtad 2021-10-15
3. Classes, Mozilla developer network,  
   <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes?retiredLocale=sv-SE>)  
   Hämtad 2021-10-15
4. Higher Order Functions, Eloquent JavaScript  
   <https://eloquentjavascript.net/05_higher_order.html>  
   Hämtad 2021-10-15
5. Understanding Higher Order Functions, Medium  
   <https://medium.com/@hiddendreamz7/understanding-higher-order-function-foreach-map-filter-reduce-in-javascript-c60c07963b23>Hämtad 2021-10-15
6. JavaScript Spread Operator, Programiz  
   <https://www.programiz.com/javascript/spread-operator>  
   Hämtad 2021-10-18
7. JavaScript Destructuring, Programiz  
   <https://www.programiz.com/javascript/destructuring-assignment>Hämtad 2021-10-18
8. JavaScript Arrow Function, Programiz  
   <https://www.programiz.com/javascript/arrow-function>Hämtad 2021-10-18
9. JavaScript Fetch API Tutorial, FreeCodeCamp  
   <https://www.freecodecamp.org/news/javascript-fetch-api-tutorial-with-js-fetch-post-and-header-examples/>Hämtad 2021-10-18
10. JavaScript: Concept of Promise, DEV  
    <https://dev.to/swarnaliroy94/javascript-concept-of-promise-3ijb>   
    2021-10-18
11. TypeScript, a beginner’s guide, Medium  
    <https://medium.com/jspoint/typescript-a-beginners-guide-6956fe8bcf9e>  
    Hämtad 2021-10-20
12. TypeScriptlang.org  
    [https://www.typescriptlang.org/](https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html)  
    Hämtad 2021-10-20
13. What is Babel?, Babel  
    <https://babeljs.io/docs/en/>Hämtad 2021-10-23