

# Introduktion till JavaScript

Mikael Svahnberg\*

2024-03-26

## 1 Introduktion

- Ett tolkat språk med rötter i funktionell programmering: **JavaScript**
- JavaScript som server-språk: **node.js**
- Ett rikt ekosystem: **npm**

Vi har väntat med detta för att kunna utveckla inuti en container

- Vanlig målplattform för node.js - applikationer idag
- Vi slipper installera mer programvara

## 2 Introduktion till JavaScript

- Utvecklades en gång i tiden för att kunna köra mindre program på webbsidor
- Följer *ECMAScript*-standard
- Är *Interpreterat* / tolkat eller *Just-in-Time*-kompilerat
- Är *Dynamiskt Typat* (eller svagt typat)
- Implementerar *prototyp-baserad objektorientering* (kommer till detta)
- Har *första-klass funktioner* (kommer till detta också)
- Lånar många drag från *funktionella programspråk*
- Kan köras i webbläsare, eller som serverplattform (t.ex. med node.js)

## 3 Interpreterat språk

- Behöver inte kompileras
- Ändringar i koden kan slå igenom direkt
- Har ofta en *REPL* för snabbare utveckling/testning

– Read-Eval-Print-Loop

---

\*Mikael.Svahnberg@bth.se

## 4 Dynamisk Typning

- Så kallad *Duck-Typing*
  - “If it walks like a duck and it quacks like a duck, then it must be a duck”
  - Variabler kan innehålla vad som helst, och kan byta typ av innehåll när som helst

```
a = 1;           // 1
a = a + 1.2      // 2.2
a = 'hello';     // 'hello'
a = { name: 'Pinkie Pie' }; // [object Object]
console.log(a.name); // 'pinkie pie'
console.log(a.handle); // undefined
a = Math.sqrt(-1); // NaN
a.sort();         // Uncaught TypeError: a.sort is not a function
a = [3,2,5];      // [3, 2, 5]
a.sort();         // [2, 3, 5]
a = function(x) { return x+1; } // [Function: a] // funktioner är också objekt
a(1);            // 2
```

## 5 Prototyp-baserad objektorientering

- Behöver inte skriva en klass, som i Java eller C++
- Räcker att ha ett *objekt*, sedan kan man lägga till metoder efter hand
- Skapa ett nytt objekt ↔ kлона ett existerande objekt

Min åsikt:

- Det här är ibland användbart när man bygger en datamodell
- Skapar huvudvärk för underhåll
  - Man måste leta för att hitta var metoder läggs till (t.ex. för att fixa buggar)
  - Man vet inte om det funkar förrän man kör programmet
  - Man måste vara **väldigt** noga med hur man lägger in ny data så att man inte råkar stoppa in en ny funktion.
    - \* Tänk om den metoden kommer från ett öppet webgränssnitt?

Please enter your name:

Bobby', save: function() console.log('Well, this is stupid'); '

▲

▼

(Lyckligtvis tror jag inte just detta kommer funka, men det illustrerar poängen)

## 6 Klasser

- Lyckligtvis stöds också klasser nuförtiden:

```
class ClassName {
  memberAttribute = 12;

  constructor() {
    this.#someData=42;
  }

  #privateFunction(param1, param2='default value') {
  }

  publicFunction() {
    this.#someData += this.memberAttribute;
    this.#privateFunction(13, 14);
  }
}

let someObject = new ClassName();
```

## 7 Funktioner

```
firstGreet('Greg');          // Functions can be called before declared (so called "Hoisting")

function firstGreet(name) {
  console.log('Hello', name);
  return 'something';        // Returns work as usual
}

function greet(name, name2=null) { // Functions can have default values on parameters
  let str = 'Hello ' + name;
  if (name2) {
    str += ' and ' + name2;
  }
  console.log(str);          // No return statement means it returns undefined
}

greet('Joe');                // Let the second parameter use the default value
greet('Jack', 'Jill');
```

## 8 Moduler

- Dela upp applikationen i hanterbara komponenter
  - Återanvänd moduler mellan olika delprojekt
  - En fil/modul per klass – lättare att underhålla
- Tre olika typer

**Core** Inbyggda i språkmiljön, t.ex. `http`, `path`, `fs`, `querystring`, `util`,  
 ...  
 – `var coreMod = require('module-name');`  
**Local** De moduler som du själv skriver  
 – `var localMod = require('my-module.js');`  
**Third Party** Moduler som andra har skrivit (mer om detta senare)  
 – `var thirdPartyMod = require('module-name');`

## 9 Lokala Moduler

- *Antingen* en separat fil
- *Eller* en separat underkatalog
  - `require('./module-subdirectory')`
  - Node.js letar efter en `package.json` eller `index.js`
  - `index.js` bör då mest “knyta ihop” resten av vad som finns i underkatalogen
  - I själva verket är varje fil fortfarande sin egen modul
    - \* man har bara samlat några moduler under ett gemensamt namn.

## 10 Skriva en Modul

- En modul kan bestå av en blandning av
  - data (som variabler eller konstanter)
  - funktioner
  - klasser (helst bara en)
- Man använder variabeln `module.exports` för att bestämma vad som skall vara tillgängligt.
  - Man kan bygga på `module.exports` efter hand eller en gång i slutet.
  - Man föredrar en `module.exports` i slutet av filen.

```
module.exports.someFunction = function() { /* ... */ }
module.exports.someOtherFunction = function() { /* ... */ }
```

```
function aThirdFunction() { /* ... */ } // This one is not exported, only usable inside th
function aFourthFunction() { /* ... */ }
function aFifthFunction() { /* ... */ }
```

```
// eller
module.exports = { aFourthFunction, aFifthFunction };
```

- Det finns också en ECMAScript-6-variant:
  - `export {exported1, exported2};`
  - `import {exported1, exported2 as exp2} from 'module-name'`

## 11 Tredjeparts-moduler: NPM

- NPM (Node Package Manager) är en paket-hanterare för node.js
  - Kommer med när man installerar node.js
  - Stort antal färdiga tredjepartsmoduler: <https://www.npmjs.com/>
- Att utveckla mjukvara idag handlar om att använda rätt tredjepartsbibliotek
- Se upp!
  - Varje paket man använder är ett potentiellt säkerhetshål
  - Varje paket kan ha sin egen utvecklingslicens
  - Ibland finns funktionaliteten inbyggd i språket
    - \* Omkring år 2021 gick halva internet ner eftersom utvecklaren av paketet `whitespace` blev sur.
    - \* I princip gjorde paketet samma sak som den inbyggda funktionen `trim()`
  - $\sum$  det är lika viktigt att känna till vad som redan finns!

## 12 Installera paket med NPM

- *Om man har npm installerat lokalt* installerar man nya paket med `npm install paket-namn`
  - Läger till paketet i filen `package.json`
  - Installerar paketet i katalogen `node-modules`
  - Installerar eventuella andra paket som det nya paketet kräver
    - \* Installerar eventuella paket som *de* i sin tur kräver.
- *Om man utvecklar i en container* är det lite krångligare
  - Man har inte npm installerat lokalt
  - Man kan manuellt lägga till paketet i filen `package.json`
  - Sedan *måste* man bygga om sin container-image och starta om applikationen!
    - \* Troligen finns där en rad i Dockerfile: `RUN npm install`

## 13 package.json

- Definierar ett projekt/ett paket. Följande *kan* beskrivas (bland annat):
  - name** Namn på projektet
  - version** Versionsnummer
  - description** Kort beskrivning av projektet
  - keywords** Nyckelord för att göra det mer sökbart

**author** Namn och kontaktinfo på projektets ägare.

**license** Projektets licens

**homepage** Länk till projektets hemsida

**bugs** Länk till bug-databasen

**main** Om du utvecklar ett paket så är det den här filen som man skall starta med. (Default 'index.js')

**repository** Länk till projektets utvecklings-sida

**scripts** Olika sätt att starta / testa projektet.

**dependencies** Vilka andra paket behövs för att kunna använda det här paketet.

**devDependencies** Om man har beroenden som är viktigare för utveckling *av* paketet snarare än *med*

- Om man startar ett projekt med `npm init` får man följande fält:
  - name, version, description, main, scripts, author, och license

## 14 Semver: Paket-version

- Man följer vad som kallas för *Semantic Versioning*, eller *semver*
  - <https://semver.org/>
- `<major>.<minor>.<patch>`, ex. 1.2.34

Major Inkompatibla förändringar i gränssnitt

Minor Ny funktionalitet som är bakåtkompatibel med den redan existerande produkten

Patch Bakåtkompatibla bugfixar

I don't know what the middle number does, but I increase the left when I break something on purpose, and I increase the right when I fix something that I accidentally broke.

- I `package.json` anger man hur exakt man behöver en viss version av sina dependencies:

**version** Exakt version, e.g. 1.2.3

**>version** Högre än version

**>=version** Minst version

**<version** Mindre än version

**<=version** Max version

**1.2.x** major och minor låst, men kan ta vilken patch som helst.

**1.x** major låst, kan ta vilken minor och patch som helst.

**~version** "Ungefär ekvivalent" till version

**~1.2.3** Minst version 1.2.3, men vilken patch som helst större än 3

**~1.2** Samma som 1.2.x

**~1** Samma som 1.x

**^version** "Kompatibel med" version; tillåt alla förändringar upp till den vänstraste icke-noll versionsnummret.

**^1.2.3**  $\geq 1.2.3 < 2.0.0$

**^0.2.3**  $\geq 0.2.3 < 0.3.0$

- Vi vill låsa oss ganska så hårt, för annars riskerar vår produkt gå sönder för att någon annan skapar en ny version.
  - Vanligast vill vi låsa oss till **^version**, e.g. **~1.2.3**

## 15 Mer om package.json

- Den allra minsta **package.json** innehåller
  - information om vilka beroenden som applikationen har (**dependencies**; installeras med **npm install**)
    - \* Dessa beroenden kan du lätt hitta på <https://www.npmjs.com/>
  - Kommandon för att köra programmet (**scripts** ; körs sedan med **npm run <scriptname>**)

```
{
  "name": "Exempel på en minimal package.json",
  "scripts": {
    "start": "node src/index.js",
    "test": "mocha",
    "dev": "nodemon src/index.js"
    // Kan fylla på med fler kommandon här (eller ta bort någon)
  },
  "dependencies": {
    "express": "^4.18.1",    // Vad betyder detta? Vilka versioner är tillåtna?
    "node-fetch": "^2.6.7",
    "mocha": "^10.4.0"
  }
}
```

## 16 JavaScript som ett funktionellt programspråk

Bakgrund – Olika *programmeringsparadigmer*

**Imperativ** beskriv steg för steg vad datorn skall göra

**Procedur-orienterad** Olika procedurer (funktioner) som kan anropa varandra.

**Objekt-orienterad** Data och beteende (procedurer) samlas i *objekt*

**Deklarativ** beskriv det önskade resultatet

**Funktionell** Det önskade resultatet beskrivs som en serie funktioner som utvärderas

**Logisk** Beskriv alla möjliga och alla kända relationer, kan sedan ställa frågor mot dessa

**Reaktiv** Fokusera på dataflöde och att förändringar skall propagera

De vanligaste språken vi använder idag har sitt ursprung i Cobol och ALGOL  
== procedurorienterade.

Parallellt levde funktionella språk som *lisp* vidare

Idag lånar många procedurorienterade språk in idéer från funktionella språk

## 17 Funktionella drag

- Funktioner är första klassens medborgare
  - De har namn
  - De är objekt som kan skickas som parametrar och returvärden
  - Som objekt kan de också vara anonyma vid behov (funktioner är också data)
- *Data* och *samlingar av data* är viktiga
  - Data processas genom att man tillämpar en serie funktioner
  - Har speciella funktioner för att arbeta mot samlingar
  - Använd dessa hellre än iterationer

## 18 Exempel imperativ vs funktionell

```
// ex 1
for (int i = 0; i < 10; i++) {
    myArray[i] = myArray[i] + 10;
}
```

```
// ex 2
for (int elem : myArray) {
    cout << elem << endl;
}
```

```
// ex 1
myArray = myArray.map( function(elem) { return elem+10; });
```

```
// ex 2
myArray.forEach( console.log );
```



## 19 Anonyma Funktioner

- *lambda-funktioner*
- Eftersom funktioner är objekt så kan man ange dem som en parameter till en funktion.
- Vanliga mönster:
  - “Gör detta och anropa `callback()` när du är klar”
  - “Tillämpa funktionen `funkis()` på alla element i samlingen”
- Det är så vanligt att man har förenklade skrivsätt:

```
let arr = [1,2,3,4,5,6,7,8,9,10];
arr.every( elem => elem<20 );           // true ; alla element är mindre än 20
arr.filter( (elem) => 0==elem%2 );      // [2,4,6,8,10]
arr.find( elem => { if (0==elem%5) {    // kan ha en hel funktionskropp
  return true;
} else {
  return false;
}} );                                   // 5 (första elementet där funktionen returnerar true)
```

```
arr.forEach( elem => console.log(elem)); // Skriver ut elementen men returnerar "undefined"
arr.forEach( console.log );             // Kan använda existerande metoder direkt, behöver inte
arr.map( elem => 2*elem );               // returnerar [2,4,6,8,10,12,14,16,18,20], men arr är oförändrad
arr.reduce(
  (accumulator, elem) => accumulator += elem, // två parametrar, måste ha parenteser runt
  0);                                         // 0 är startvärdet på ackumulatorn, returnerar '55'
```

## 20 Händelsestyrning

- Asynkron programmering
  - Saker sker inte i en deterministisk ordning
  - Man låser funktioner till vissa händelser (*Event*)
  - Antingen genom `EventEmitter`, eller genom `callbacks`

```
// ex 1 (EventEmitter)
const EventEmitter = require('node:events');
const myEventEmitter = new EventEmitter();

myEventEmitter.on('message', () => { console.log('Received message!'); });
myEventEmitter.emit('not-the-message'); // inget händer, finns ingen som lyssnar på 'not-the-message'
myEventEmitter.emit('message');
```

```
// ex 2 (Callback)
const fs = Require('node:fs');
fs.readFile('./someFile.txt', fileRead); // the method fileRead is only called once the file is read
console.log('This will be printed immediately'); // fs.readFile will continue working in the background
```

```
function fileRead(err, data) {
  if (err) { console.log('could not read file'); return; }
  // Do something with the contents of the file (in the parameter 'data')
}
```

## 21 Callbacks: Flera trådar fast ändå inte

- JavaScript-världen blev förälskade i lambda-funktioner och callbacks
- De överanvände det utan att ta lärdom av 60+ år av erfarenheter från lisp- och fp-världen.
- Kallas för *callback hell*

```
readFile(filename, (result) => {
  processData(result, (result2) => {
    filterWithDatabase(result2, (result3) => {
      // och så vidare
    });
  });
});
```

- Det här ser fortfarande ganska så rent ut, fyll nu på varje nivå med 10-15 rader kod till!
- Vi kan snygga till det genom att använda namngivna funktioner men det är fortfarande svårt att läsa.

## 22 Promises och async/await

- en *Promise* är javascripts lösning för att undvika callbacks.
- *async/await* är mest ett enklare sätt att skriva promise-kod

```
// ex 1 (med Promise)
let result = readFile(filename)
  .then( result => processData(result) )
  .then( result => filterWithDatabase(result) )
  .catch( err => { /* ... */ } )
  .finally( result => { /* ... */ } );
```

```
// ex 2 (med async/await)
async function doTheThing() {
  let fileData = await readFile(filename); // med await så väntar vi ut den här processen
  console.log(fileData); // Därför kan vi nu anta att variabeln innehåller det
  let processedData = await processData(fileData);
  let filteredData = await filterWithDatabase(processedData);
  return filteredData;
}
```

```
doTheThing().then( console.log ); // Vi kan nu hänga på ännu fler saker som ska
console.log('This may be executed before doTheThing returns.');
```

- Det finns mycket mer att säga om Promises
  - t.ex. `Promise.resolve()` och `Promise.reject()`
  - Hur man skapar sin Promise till att börja med
  - ...

## 23 Mer Funktionell Programmering

Andra saker som man *kan* låna från funktionell programmering:

**Immutable Data** Allt är konstanter, det går inte ändra värdet på en variabel.

**First class functions** Funktioner är också objekt som kan hanteras på samma sätt som andra variabler

**Homoiconic** Kod är också data, data kan vara kod.

Och *Pure Functions*:

- Funktioner har inga sidoeffekter (de ger ett svar men uppdaterar ingenting annat).
- Svaret från en funktion är alltid samma, givet samma indata.
- Man kan anropa funktioner i vilken ordning som helst och får alltid samma svar.
- (Är egentligen en konsekvens av *immutable data*)

Fördelar:

- Lättare att förstå koden om ingenting annat än returvärdet från en funktion påverkas
- Lättare att skrivatrådade program

Med *god disciplin* kan man programmera på det här viset i JavaScript också.

## 24 Sammanfattning

- JavaScript används mycket i webbläsaren
- ... men i ökande utsträckning också som server-språk
  - t.ex. med `node.js` eller nyare varianter som `Bun`
  - med pakethanteraren *NPM* kommer en rik samling med färdiga paket som man kan använda i sin applikation.
- Programspråket i sig själv är ganska standard C-liknande syntax
- Klasser stöds men är inte kritiska

- JavaScript är *dynamiskt typat*, en variabel kan byta typ av innehåll.
- Varje fil är en `modul` som kan ha privata och publika delar.
- En del användbara begrepp lånade från funktionell programmering
  - Fokus på *data* och *samlingar av data*
  - Lambda-funktioner, funktioner som data
  - *Pure functions* uppmuntras i många situationer.

## 25 Nästa Föreläsning

- Mer om applikationsutveckling i JavaScript
- Skrivbordsapplikationer
- Klient-Server-applikationer
- Testning
- Curses, ett klassiskt ramverk för textbaserade gränssnitt: [https://en.wikipedia.org/wiki/Curses\\_%28programming\\_library%29](https://en.wikipedia.org/wiki/Curses_%28programming_library%29)
- Introduktion till testramverket Mocha för JavaScript: <https://javascript.info/testing-mocha>

## 26 Övning: Kom igång med JavaScript

### 26.1 Introduktion till Övningen

- Den här gången skall vi börja från grunden med ett helt nytt projekt.
- Du kanske vill skapa projektet på din git-server först och kлона det därifrån
  - Om inte, starta åtminstone i en ny katalog med `git init`
- Projektet går ut på att man får fylla i förnamn och efternamn på en websida, och få en hälsning tillbaka.
- Som en del av projektet skall du *minst* skapa följande
  1. En `Dockerfile` som
    - installerar `nodemon`,
    - installerar alla övriga beroenden från `package.json`, och
    - startar applikationen med `ENTRYPOINT ["npm", "run", "dev"]`.
  2. En `package.json` som minst:
    - deklarerar `Express ^4.19.1` som ett beroende
    - har ett `dev` - script som startar applikationen med hjälp av `nodemon`.
  3. En fil `src/index.js` som startar upp en express-webapplikation med två routes

- GET / som levererar en sida enligt nedan
  - GET /greet som lägger till en rad `Hello, Firstname Lastname!`.
4. En fil och en klass `src/person.js` som
- representerar en person med `firstname()`, `lastname()`, och `fullname()`.
  - Lagrar namnen med versal första bokstav (ex lagras "john" som "John")
  - Har en metod `greet()` som returnerar `this.fullname()`.
  - Glöm inte `module.exports = Person`
5. (gärna) en `makefile` med två regler:
- en `build` (`docker build . -t namegreeter`)
  - en `run` (`docker run -it -p8080:3000 -w /app -v ./src:/app/src namegreeter`)

Please Enter your name:

Firstname

Lastname

## 26.2 Spara Hälsningar

- Nästa steg är att lägga till en sida till `GET /list` som visar alla personer man tidigare hälsat på

Att göra

1. Lägg till en array `previousGreetings` i `index.js`, som du sparar dina `Person` - objekt i.
2. Lägg till en route till `GET /list` i `index.js` som visar alla personer från din `previousGreetings`.

## 26.3 Räkna Hälsningar

- För varje person man hälsar på, kolla i `previousGreetings` om du har hälsat på den personen innan (antag att om "Firstname Lastname" är samma så är det samma person)
- Lägg till en räknare i `Person`-klassen som ökas varje gång man hälsar på den personen.

## 26.4 Familjerelationer

- Om bara efternamnet stämmer, (och inte förnamnet) så är det en släkting.
- Lägg till metoden `addRelative(aPerson)` i din `Person`-klass.
  - Dubbelkolla (för säkerhets skull) så att personen inte redan är listad som en släkting
- Fixa `GET /list` så att alla släktingar listas för varje person.
- Fixa `GET /list` så att namnen skrivs ut i bokstavsordning baserat på efternamnet.

## 26.5 Sammanfattning

- Du har nu:
  - skrivit en enkel web-applikation i en container
  - skrivit en klass i Javascript.
  - Lagt till metoder i klassen.
  - Sparat objekt i samlingar och hämtat dem därifrån.