 **Closures og Callback functions**  
  
I Javascript findes der særligt begreb der hedder Closures. Det dækker over den egenskab, at en funktion kan have adgang til det ydre scope (parent scope, det scope hvor i funktionen er defineret) selv efter at det ydre scope (parent function, den funktion hvor i funktionen er defineret) er lukket.  
  
**Eksempel:**

function greet(whattotsay) {

return function(name) {

console.log(whattosay + ' ' + name);

}

}

Funktionen *greet( )* er en funktion der returnerer en ny anonym funktion. Den anonyme funktions definition kaldes et “function expression”, da den returnerer et nyt funktions objekt.

Når greet kaldes returneres en funktion, som igen kan kaldes - det gøres med de ekstra parenteser ( ):

greet('Hi')('Tony'); //output: Hi Tony

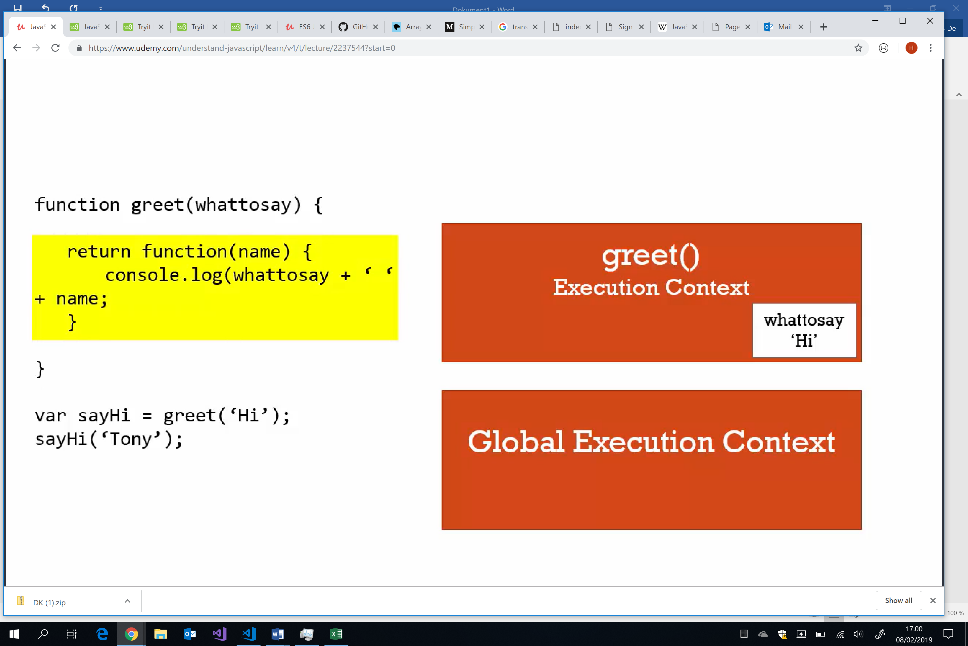
Vi kan også tildele funktionen der returneres fra greet til en variable og så kalde funktionen senere:

var sayHi = greet('Hi');

sayHi('Tony'); //output: Hi Tony

Spørgsmålet er: hvordan ved sayHi( ) funktionen hvad whatToSay variablen indeholder? – WhatToSay variablen er jo poppet af stakken da greet( ) – funktionen er færdigafviklet.  
Dette er pga. Closures!

1)



1) Når koden afvikles oprettes fra start en Global Execution Context

2) Når koden var sayHi = greet(’Hi’) køres oprettes en Execution Context for greet.

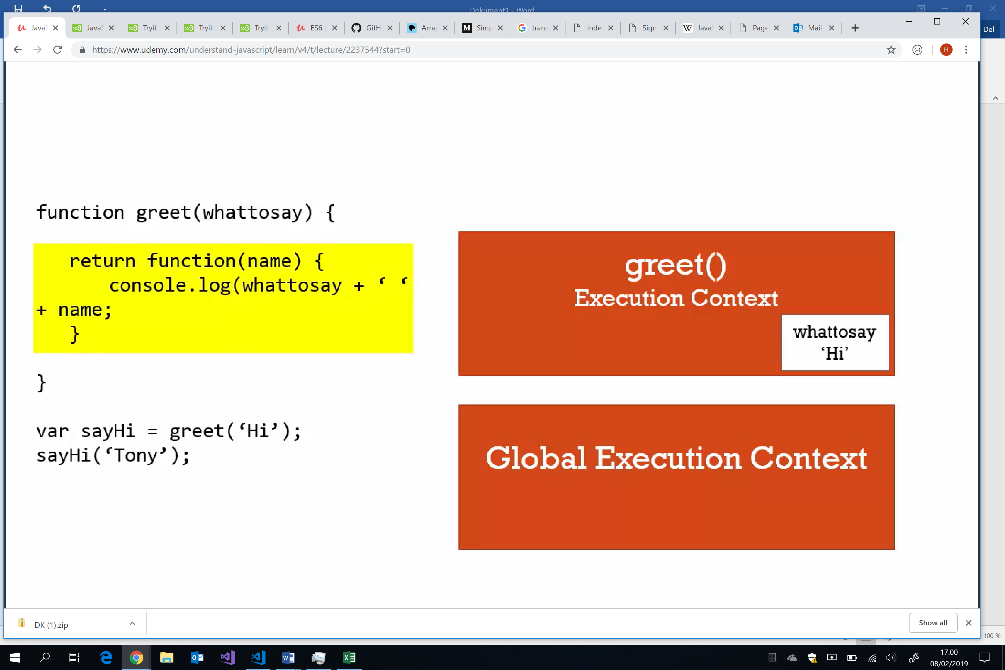
3) Parameteren whattosay oprettes i Execution Context og initialiseres med argumentværdien ’Hi’  
  
4) Der oprettes et nyt funktions objekt:

function(name) {

console.log(whattosay + ' ' + name); }

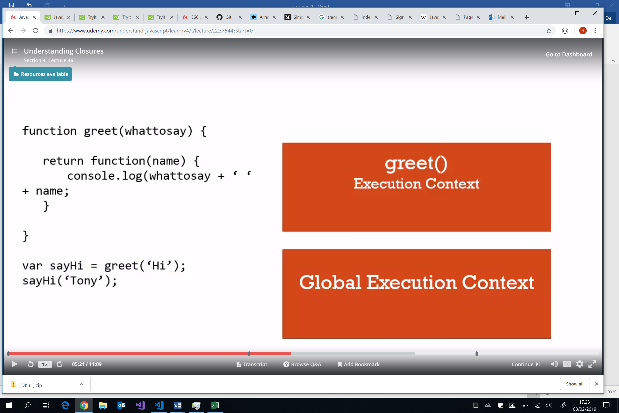
der returners.

2)



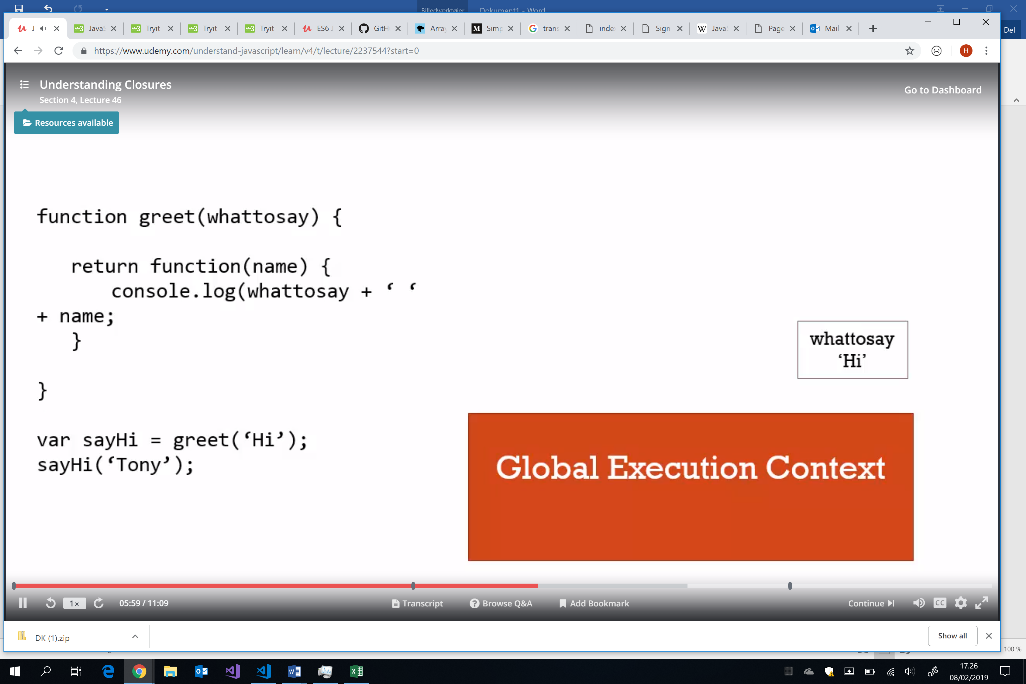
3)

5)



5) Når greet( ) har returneret funktions objektet poppes Excecution Context af stakken

6) Normalt frigives hukommelsen når et Execution Context poppes af stakken, men her bevares det. whattosay fjernes ikke af Garbage Collectoren, da der stadigt findes en reference til variablen



6)

7) vi er inde i Global Execution Context når koden sayHi(’Tony’) eksekveres.

8) der oprettes et nyt Execution Context til den anonyme funktion, der kaldes gennem sayHi.

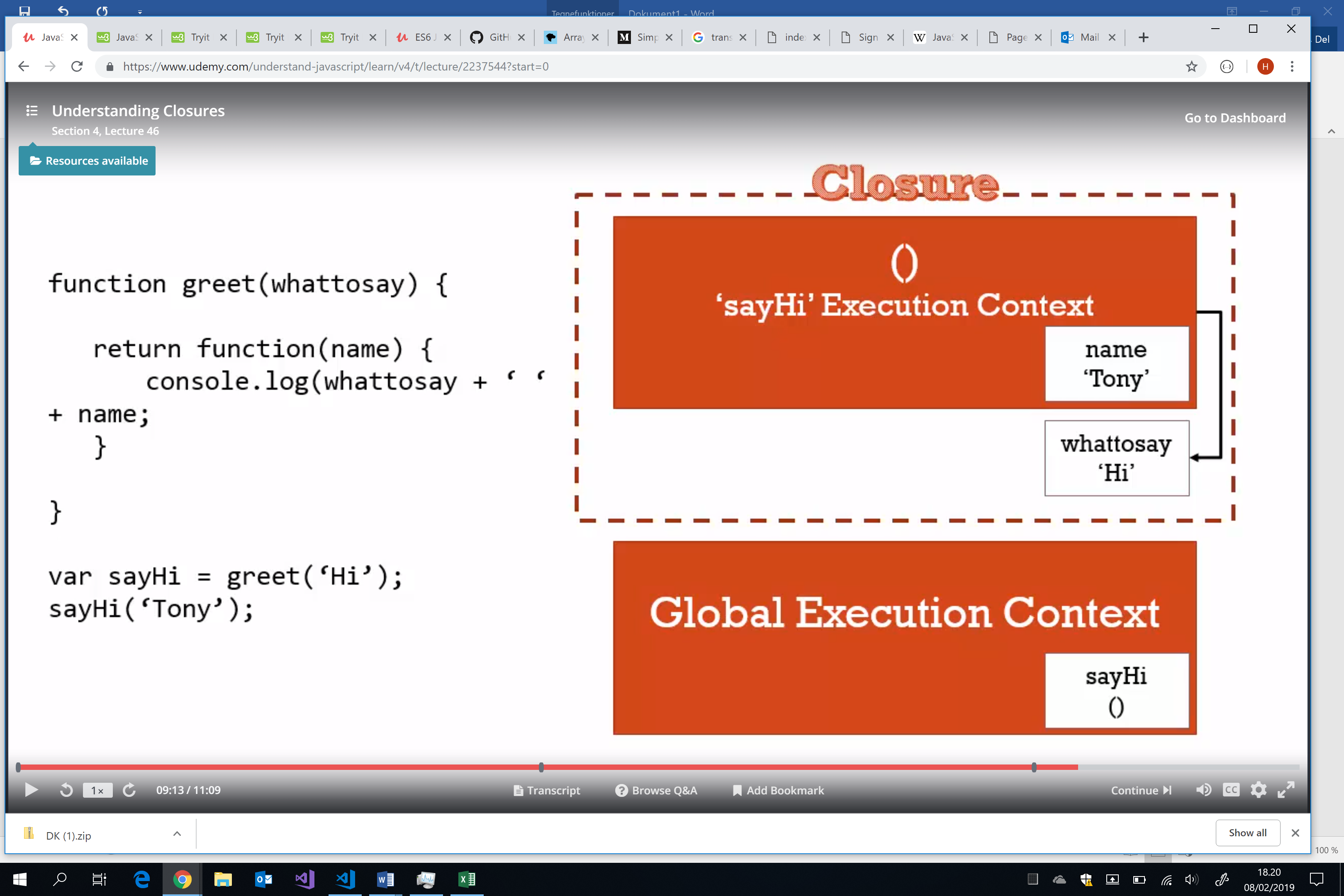
9) variablen name indsættes med værdien ’Tony’

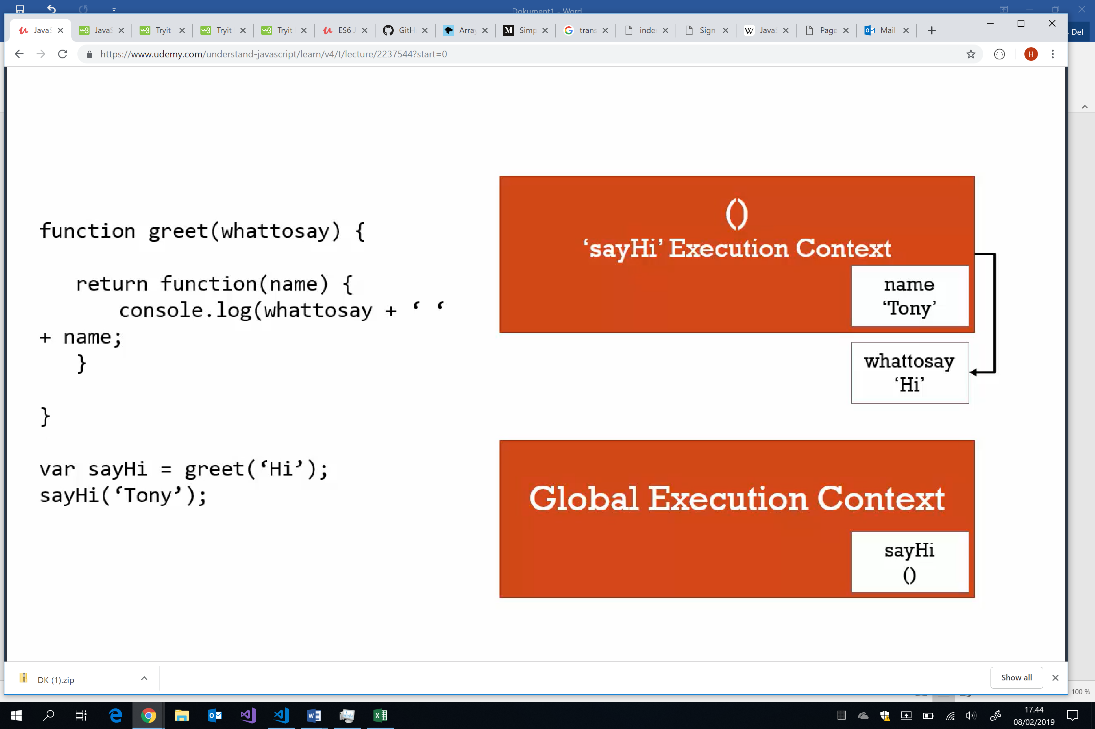
10) Når koden:

console.log(whattosay + ' ' + name);

vil JS Engine lede efter whattosay variable gennem ”scope chain”, ved at benytte Outer Environment Reference, der referere til hvor funktions objektet blev skabt.

11) Selv om Execution Context for greet ikke lever mere, har ’sayHi’ Execution Context en reference til whattosay i hukommelsen.



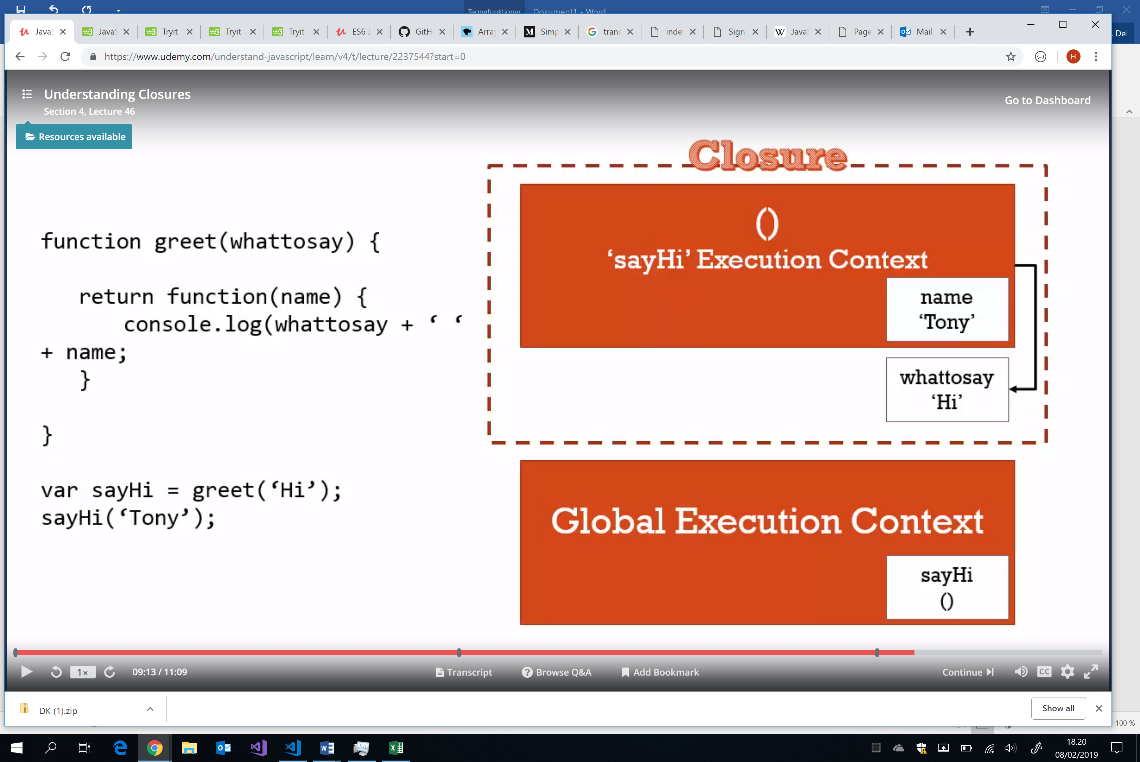


8)

9)

11)

7)



12) JS Engine sikre at funktionen stadigt kan gå ned af ”scope chain” og finde variable mv fra Execution Context selv om det er poppet af stakken.  
  
Man siger at Execution Context har ”Closed in” de ydre variable som den ellers have adgang til. Det er det fænomen der kaldes Closure.

**Callback functions**  
En callback function er en funktion der gives som argument til en anden funktion, for at kunne blive kaldt når den anden funktion er færdig/afsluttet.  
  
”So the function you call (ie invoke) ’calls back’ by calling the function you gave it, when it finishes”.  
  
**Eksempel:**Følgende lille eksempel viser en “template” for hvordan callbacks virker:

function tellMeWhenDone(callback){

// .... some work

callback();

}

tellMeWhenDone(function(){console.log("Im' done!")});

tellMeWhenDone(function(){console.log("All done!")});