SCOPE, CLOSURES & CONTEXT

LITTERATUR

You Don't Know JavaScript: Scopes & Closure

Djupgående men nyttig

Mozilla Developer Network: JavaScript Reference

Trist men bra

.MAP .FILTER .REDUCE

FUNKTIONELL PROGRAMMERING

Immutability: inte ändra på datastrukturer bara kopiera

Bygg på från föregående **return**

Pure functions

No side effects

Ref: What is Functional Programming

.MAP

Vi mappar om arrayen. Returnerar en ny array baserad på den gamla arrayen.

```
var newArray = countries
  .map(function(country){
    return country.name;
})
```

```
var timestwo = numbers.
  map(function(number){
    return number * 2;
}
console.log(timestwo);
```

.FILTER

Returnerar en filtrerad array baserad på vårt villkor.

```
var newArray = countries
  .filter(function(country){
    return country.population > 5000000;
}
```

Om villkoret ställer kommer objektet att sparas i den nya arrayen

Funktionen som **filter()** tar emot bestämmer vad filter ska göra.

.REDUCE

Reducerar innehållet i arrayen baserat på vårt vilkor.

DEN FÖRVIRRANDE DELEN

```
countries.map(country, index, array){
  return country.map;
}
```

country : värdet på varje index

---

index : varje index helt enkelt

---

array : själva arrayen vi loopar igenom

DEN FÖRVIRRANDE DELEN: DEL 2

```
countries.
  reduce(function(pop, country, index, array)
  return pop += country.population;
}, 0)
```

pop: det samlade värdet efter varje iteration

---

country: värdet på varje index

0: Vi säger åt reduce att startvärdet ska vara 0.

```
countries.
  reduce(function(semla, obama){
    return semla + obama.population;
}, 0)
```

Som en tumregel spelar det nästan aldrig någon roll vad du döper dina parametrar till, bara vilken ordning de är i.

Gäller det mesta i JavaScript

SCOPES & CLOSURES

Context, hoisting & shadowing

SCOPE

Beroende på vilket scope vårt kodblock körs i så kommer koden att ha tillgång till olika variabler. Scope bestämmer vad som är tillgängligt var.

Men vi bestämmer i vilket scope vi lägger koden i.

Om vi lägger en variabel i en funktion så har vi valt att variabeln ska tillhöra det scopet och får ta konsekvenserna.

PRINCIPE OF LEAST PRIVILEGE

Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.

Ref: Principle of Least Privilege

RUNTIME

Hur koden väl körs bestäms under **Runtime**: när koden körs

Variabler och funktioner är inte strikt bundna till ett objekt

---

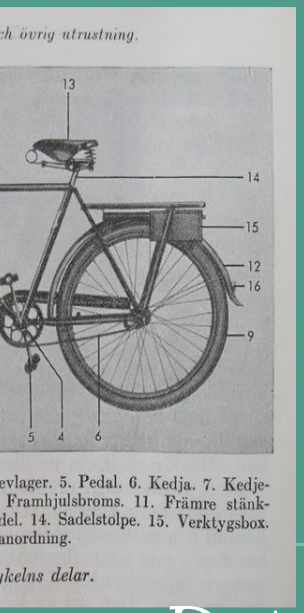
Hur funktioner körs och hur värden konverteras bestäms vid runtime. Ska värdet bli **"0"** eller **0**?

LEXICAL SCOPE

Det scope som en variabel eller funktion ligger i kallas **lexical scope**

På det ställe som variablen är på i kodstrukturen.

En funktion behöver inte nödvändigtvis vara bunden till sitt lexical scope.



LEXICAL SCOPE

Jag äger en cykel, det är min cykel

Det hindrar inte dig från att cykla på min cykel. Du kan cykla och jag kan låta dig låna min cykel.

Men cykeln är min, den står utanför min lägenhet.

```
function foo(a) {
    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

1

2

3

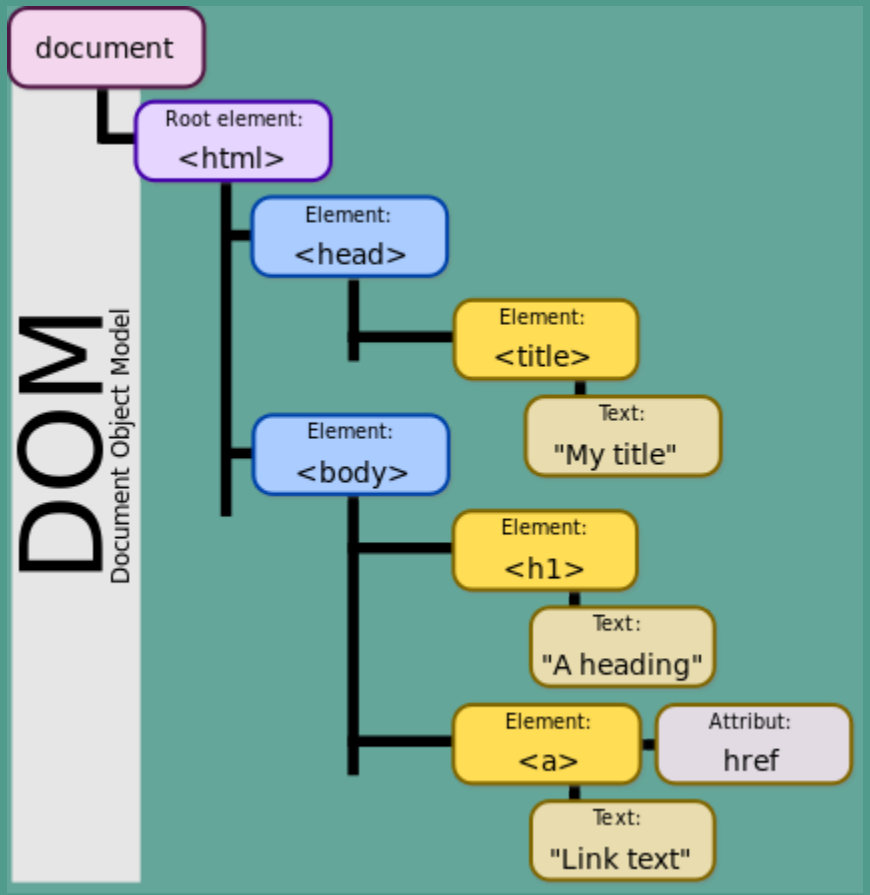
CONTEXT

När man pratar om context brukar man oftast prata om vad exakt this är.

Vår kod körs alltid i browsern, alltså i webbläsaren

---

this i global är **window**


DOM
Document Object Model
document
Root element:
<html>
Element:
<head>
Element:
<title>
Text:
"My title"
Element:
<body>
Element:
<h1>
Text:
"A heading"
Element:
<a>
Attribut:
href
Text:
"Link text"

TUMREGEL

Scope är var variabeln/funktionen ligger

Context är vilket objekt variabeln/funktionen tillhör.

VAR VS. LET

I JavaScript har vi i princip bara **Global** och **function** scope

---

let används för att skapa block-baserade variabler

Men vad är ett **block**

Ref: MDN - let

LET

```
for(var i = 0; i < 10; i++){
  console.log("i exists outside of loop");
}

for(let i = 0; i < 10; i++){
  console.log("i only exists in loop")
}
```

BLOCK SCOPE

if/else/else if

for/while/do while

{}

Använd **let** så mycket som möjligt.

Använd när du vill skapa tillfälliga variabler

let hoistas inte

HOISTING

Variabler och **Function declarations** läggs högst upp i dess nuvarande scope

---

Undantag: **let**

Funktioner och variabler är tillgängliga för hela scopet

Undatag: **Function Expression**

DANGER DANGER

Variablen hoistas men det är inte säkert att värdet gör det

```
var func = function(){}
```

SHADOWING

När en variabel skuggar en annan

when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope.

Ref: Wiki - Variable Shadowing

```
var data = 42;
  function funkis(){
    var data = 0;
    console.log(data);
}
```

Den inre variablen skuggar den yttre

Bra & Dålig grej: ofta dålig

CONST

```
const PI = 3.14;
```

pi är alltid detsamma

Använd när något värde inte ska ändras

Skriv med stora bokstäver för att förtydliga, men inget måste.

Ref: MDN - const

VAD SKA VARA KONSTANT?

```
const sayHello = function(){
  return "Hello";
}
```

Semantisk skillnad: vi säger att denna funktion inte ska skrivas över.

Särskiljer den från vanliga variabler.

ARROW FUNCTION =>

```
var hello = function(){
  return "Hello";
};

var hello = () => {
  return "Hello";
};

var hello = () => "Hello"
```

ARROW FUNCTION =>

NÄSTAN samma sak som en vanlig funktion

Binder funktionen till den nuvarande kontexten (this)

Bra **OCH** dålig grej

Inte alltid en direkt ersättare för en funktion. Används oftast istället för anonyma funktioner.

Ref: Exploring ES6 - Arrow functions

() => LEXICAL SCOPE

Oftast bra när man ska köra anonyma funktioner (kallas även lambdas)

Försäkra sig om att this alltid är detsamma.

Ibland vill vi dock att this ska kunna ändras.

Ref: Exploring ES6 - Arrow functions

CALL, APPLY

Vi kan kalla på en funktion som inte är bunden till ett objekt

---

```
myFunction.call(obj, arguments);
```

```
myFunction.call(obj, [arguments]);
```

Vi kan ha fristående funktioner som kan kalla på vilket objekt vi vill.

Har vi mer generella metoder behöver de inte vara bunda till objektet.

KODEXEMPEL

CLOSURES

FREE VARIABLES

En closure är egentligen inget fuffens.

Variabler i det yttre scopet finns kvar som referens även fast funktionen har kört klart.

Funktionen "kommer ihåg" variablen och kan använda den även fast den är utom räckhåll för resten av koden.

Skapas automatiskt när vi deklarerar en funktion i en annan funktion.

Ref: MDN - Closures

Ref: MDN - Closures

PRIVATA VARIABLER

Oftast används det för att skapa privata variabler i en funktion eller objekt

Variabler som man döljer undan och kan styra hur de sätts och hämtas

Principle of Least Privilege

IIFE

Immediately-Invoked Function Expression

Function expression som kallar på sig själv

```
(x => x * 10)(2)
```

```
var invoked = (function(){
  return "Hello IIFE!";
})();
```

Observera paranteserna

Ref: A Drip of JavaScript: IIFE

IIFE används ofta för att skapa closures för att hålla privata variabler

```
function foo(){
  var bar = 'Private variable';
  return function(){
    console.log(bar);
  }
}
```

```
var innerFunction = foo();
innerFunction();
```

SAMMANFATTNING

Använd `let` när det behövs, tänk på att variabeln inte hoistas

---

Använd `const` för konstanter och funktioner

Använd `=>` så ofta som möjligt men tänk på att det kan ställa till det. `this` är alltid bundet.
