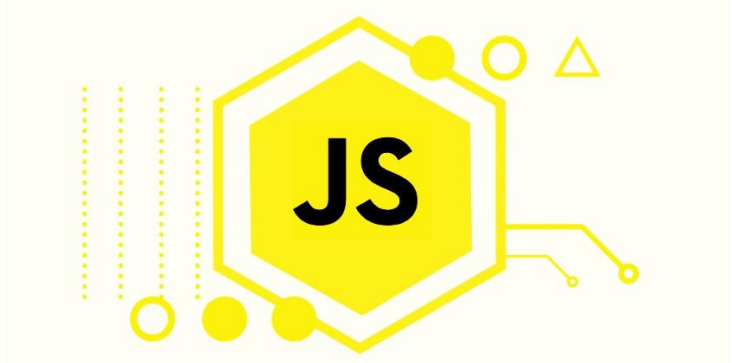


Javascript-ის კურსი

Chapter 1. შესავალი. გაცნობითი ლექცია

Chapter 2. ვისწავლით JS-ის საფუძვლებს რაც გაგვიადვილებს შემდეგი თემის გაგებას.

Chapter 3. განვიხილავთ როგორ მუშაობს JS სცენის მიღმა.

Chapter 4. განვიხილავთ JS-ს browser-თან მიმართებაში:

- რა როლი აქვს.
- რა არის DOM(document object model).
- ავაწყობთ პატარა თამაშს რათა პრაქტიკულად დავინახოთ JS-ის ძალა.

Chapter 5. სიღრმისეულად განვიხილავთ JS-ს და ვისაუბრებთ Object-სა და Function-ებზე

Chapter 6. ამ თავში, დაგროვილი ცოდნით გავაკეთებთ პრაქტიკულ ამოცანას, Budget-ის პროგრამას, რომელიც მოგვცემს ბევრ სიახლეს, სტრუქტურულად პროგრამის სწორად წერასა და პრობლემებთან ბრძოლის უნარებს.

Chapter 7. ამ თავში განვიხილავთ ESnext (esmascript, ანუ ES6/ES5 2015) ანუ ახალი თაობის JS-ს

Chapter 8. Asynchronouse JS პროცესი API(application Processing Interface)-სთან მცირედი გამოცდილებით.

Chapter 9. ამ თავში ჩვენ შევაჯამებთ ყველაფერს და უახლესი ტექნოლოგიებით გავაკეთებთ ერთ-ერთ სრულფასოვან პროექტს, სადაც გამოვიყენებთ npm buble-სა და web pack ხელსაწყოებს.

სანამ მუშაობას შევუდგებით ჩვენ დაგვჭირდება Text Editor რომელშიც ვიმუშავებთ და შევძლებთ კოდის წერას. შეგვიძლია გამოვიყენოთ Brackets ან VS code რომელიც თქვენთვის

მისაღებია. ამათ გარდა არის Atom, Sublime text და ა.შ. რაც შეეხება browser-ს გამოვიყენებთ Chrome-ს, რადგან მასში ბევრი ისეთი საშუალებაა რომელიც დეველოპერს სჭირდება და ეხმარება სრულფასოვან მუშაობაში.

Chapter 2.

2.1 მარტივი კოდის დაწერა სადაც ავხსნით inline და external სკრიპტებს შორის განსხვავებას

აქ ჩვენ გვაქვს index.html დოკუმენტი, რომელიც თავის მხრივ არის Folder-ში. Visual Studio Code-ში მოცემული ფოლდერი უნდა გავხსნათ, შესაბამისად გამოჩნდება მთლიანი ფოლდერის შიგთავსი(მარცხნივ), საიდანაც მარტივად შევძლებთ ნავიგაციას დოკუმენტებზე. მუშაობისას მე არ გამოვიყენებ გარე JS დოკუმენტს, არამედ გამოვიყენებ inline script-ს. HTML-ში inline script-ისათვის შემოგვაქვს <script> tag(ანუ სექცია) და მასში ვწერთ მარტივ კოდს JS-ის სინტაქსით console.log("hello World"); ეს ჩანაწერი კერძოდ Hello World არის ყველა პროგრამისტის პირველი შედეგი. ანალოგიურად ამ კოდის შედეგიც იგივე იქნება, მაგრამ ეს ყველაფერი chrome-ის console რეჟიმში გამოჩნდება თუ კი developer option-ს გავააქტიურებთ.

ახლა განვიხილოთ ის მეთოდი, როდესაც JS კოდი html-ში გარედან შემოდის, ანუ JS დოკუმენტის ინტეგრაცია, რომელსაც იგივე folder-ში შევქმნით, მივცემთ რაიმე სახელს (მაგ. Script.js) და მოცემულ დოკუმენტს html ჩანაწერებში <script> tag-ში დამატებით ავტომატურად მივუთითებთ. სრულფასოვანი ჩანაწერი ასე გამოიყურება : <script src="script.js"> </script>. თუ შევამოწმებთ, იგივე შედეგს მივიღებთ. ეს გზა უფრო მიღებულია რადგან HTML-ისა და JS-ის კოდი უფრო იზოლირებულია

2.2 JS-ის გაცნობა - რა არის ? სად გვხვდება ? როგორ მუშაობს ? მისი ნამდვილი სახელი ?

რა არის JS ?

Javascript არის lightweight, cross-platform, object-oriented computer programming language.

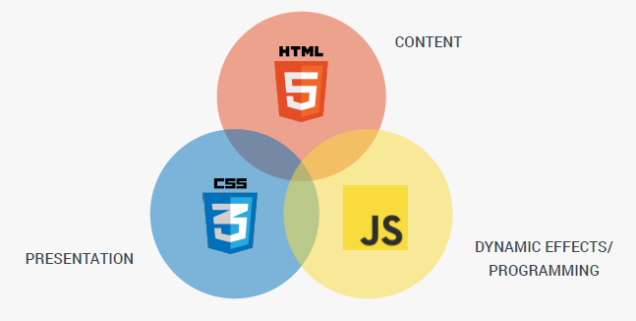
Lightweight - ანუ არ სჭირდება დიდი რაოდენობით computer memory. მას აქვს შედარებით მარტივი სინტაქსი, თავისი მახასიათებლებით.

Cross-platform - ანუ შეგვიძლია გამოვიყენოთ მრავალ განსხვავებულ platform-ზე სისტემაში და არა მხოლოდ web development-ში.

Object-oriented - ანუ ეს არის ენა, რომელიც საბაზისოდ იყენებს object-ს ჩვენ ამ ყველაფერს მთელი კურსის განმავლობაში ავხსნით.

Web ტექნოლოგიების მთავარ სამეულში შედის JS-ი html-სა css-თან ერთად. JS ტრადიციულად გამოიყენებოდა მხოლოდ browser-ის გარემოში, რაც იმას ნიშნავს, რომ ის იყო მხოლოდ client-side language. თუმცა მოგვიანებით ისეთი ტექნოლოგიის წყალობით როგორიცაა nodeJS დეველოპერებს უკვე შესაძლებლობა გვაქვს JS ასევე server side-ზეც გამოვიყენოთ. ეს ღრმა თემაა რომელსაც ამ კურსში არ განვიხილავთ, ასე რომ JS-ს აქვს ბევრი შესაძლებლობა. დღესდღეობით ჩვენ რასაც ვიყენებთ, ეს იქნება ცნობილი web page-ები თუ მობილური აპლიკაციები, მათ უკან დგას JS.

დღესდღეობით არის ბევრი framework და libraries, რომელიც დეველოპერებს ეხმარება ააგონ უფრო რთული app მეტად სწრაფად და მარტივად ვიდრე მანამდე. მე ვსაუბრობ ისეთ framework-ებზე როგორებიცაა react, angular, Jquery და ა.შ. რომლებიც არიან ძალიან პოპულარულები. ეს ყველაფერი 100%-ით შექმნილია JS-ზე. დეველოპერები სანამ ამ framework-ებს გამოიყენებენ აუცილებელია მათ სიღრმისეულად იცოდნენ JS. მაშასადამე html css და JS მუშაობს ერთად იმისათვის, რომ შევქმნათ ლამაზი, ინტერაქტიული და დინამიური web site/application.



HTML პასუხისმგებელია გვერდის შიგთავსზე, როგორიცაა ტექსტი, სურათები, ღილაკები და საერთოდ რასაც web page-ზე ვხედავთ, ეს ყველაფერი დაწერილია html-ში. ამის შემდეგ CSS არის პასუხისმგებელი იმაზე თუ როგორ გამოიყურებოდეს ეს შიგთავსი. JS არის პროგრამული ენა, რომელიც აძლევს დეველოპერს content/css-ზე მანიპულაციის საშუალებას, რაც web გვერდს უფრო დინამიურს, ინტერატიულს ხდის. მაგ შეგვიძლია:

- HTML შევადაროთ არსებით სახელს(noun-ს).
- CSS შევადაროთ ზედსართავ სახელს(adjective-ს).
- JS შევადაროთ ზმნას(verb-ს).

ეს მაგალითი უფრო მარტივად დაგანახებთ მათ როლს. ბოლოს მინდა ვთქვა რომ JS-ს აქვს განსხვავებული ვერსიები, რომლებსაც კურსის ბოლოს განვიხილავთ, მაგრამ მაინც მინდა იცოდეთ, ამ მომენტისათვის ჩვენ ES5-ის ვერსიაში ვმუშაობთ.

2.3) Variables and data types

ცვლადები არის ყველა პროგრამული ენის საბაზისო ტერმინი, მაგრამ მაინც რა არის variable ?

> **Variable - ეს არის container, რომელიც ინახავს მნიშვნელობას გარკვეული წესით. ის შეგვიძლია გამოვიყენოთ კოდში იქ, სადაც გვინდა და რამდენჯერაც გვინდა ნაცვლად ამ მნიშვნელობისა**

Variable-ს გააჩნია სახელი და მნიშვნელობა. JS-ში, როგორც ყველა პროგრამულ კოდში, ცვლადის სახელს აქვს თავისი წესები და თუ გინდათ რომ წესის გვერდის ავლით დავარქვათ სახელი მაშინ უნდა გამოვიყენოთ $-ის ნიშანი ან _ მაგ. $3a ან _3a(მარტო 3a შეცდომაა). რაც შეეხება მნიშვნელობის მიკუთვნებას, ის დამოკიდებულია Data type-ზე - Number, String, Boolean, Undefined დაNnull.

ყველაზე მთავარი გაითვალისწინეთ ის, რომ JS-ში არის Dynamic typing. ეს იმას ნიშნავს, რომ ცვლადის ტიპი დინამიურად განისაზღვრება მისი მნიშვნელობის მიხედვით. ეს ძალიან გამოყენებადია და გვიმარტივებს კოდის წერას, მაგრამ

ფრთხილად!!! ამ შემთხვევაში რთულია იპოვოთ bug და დააფიქსიროთ შეცდომა.

ასევე დაიმახსოვრეთ, პროგრამაში ცვლადის სახელად ვერ გამოიყენებთ Keyword-ებს: if, for, delete და ა.შ.

app.js
```
/*****************************
* Variables and data types
*/

var firstName = 'John';
console.log(firstName);

var lastName = 'Smith';
var age = 28;

var fullAge = true;
console.log(fullAge);

var job;
console.log(job);

job = 'Teacher';
console.log(job);

// Variable naming rules
var _3years = 3;
var johnMark = 'John and MArk';
var if = 23;
```

2.4) Variable mutation and type coercion

ამ თავში ჩვენ გავაგრძელებთ ცვლადების დახასიათებას და მინდა გითხრათ, რომ ცვლადები განიცდიან მუტაციას ანუ ცვლილებას, რაც იმას ნიშნავს რომ ცვლადის ახალი მნიშვნელობა ძველს აუვნებელყოფს. მოცემულ მაგალითში კარგად ჩანს ცვლილება და დაიმახსოვრეთ JS-ში ყველა ცვლადის დეკლარაცია იწყება var-ით.

Var job = ”teacher”;

job=”IT”;

მინდა გითხრათ რომ JS-ში როგორც სხვა პროგრამულ ენებში გვაქვს არითმეტიკული მოქმედებები: +, -, *, /, % რომლითაც ცვლადების მუტაციის მაგალითის მოყვანა უფრო მარტივია:

var x = 5;

var y = 7;

x = x + y;

არითმეტიკული მოქმედება მარჯვნივ რომ დამთავრდება, მოხდება x-ის მუტაცია, ასევე ჩვენ ვხვდებით Type Coercion, რაც იმას ნიშნავს, რომ Dynamic typing-ის საშუალებით **ერთ ცვლადში მოთავსდება სხვადასხვა ტიპის მნიშვნელობები.** ისინი იძულების წესით გაერთიანდებიან ერთ ტიპში და ეს ტიპი ხშირ შემთხვევაში String ტიპის სახეს ღებულობს (ანუ ჩანაწერის) მაგ.:

var x = 5;

var t = “7”;

console.log(x + t);

შედეგი იქნება “57” ანუ **+ ოპერაცია String type-ში აკეთებს გაერთიანებას.** ამ შემთხვევაში მოხდება Coercion (იძულება).

app.js
```
/*****************************
* Variable mutation and type coercion
*/

var firstName = 'John';
var age = 28;

// Type coercion
console.log(firstName + ' ' + age);

var job, isMarried;
job = 'teacher';
isMarried = false;

console.log(firstName + ' is a ' + age + ' year old ' + job + '. Is he married? ' + isMarried);

// Variable mutation
age = 'twenty eight';
job = 'driver';

alert(firstName + ' is a ' + age + ' year old ' + job + '. Is he married? ' + isMarried);

var lastName = prompt('What is his last Name?');
console.log(firstName + ' ' + lastName);

```

2.5) არითმეტიკული და ლოგიკური ოპერაციები

წინა ლექციებში ჩვენ ვნახეთ არითმეტიკული ოპერაციები, მაგრამ ეხლა მოდი დეტალურად ვნახოთ არითმეტიკული/ლოგიკური ოპერაციები და typeof.
არითმეტიკულში - პლიუსი, მინუსი, გამრავლება, გაყოფა (/ და %).
ლოგიკურში - ლოგიკური ნიშნები - >, <, <=, >=, ==, === სადაც 2 operand გვყავს და შემდეგი ლოგიკური კავშირი იქნება true ან false.
რაც შეეხება typeof, მას მხოლოდ ერთი operand ჭირდება და შედეგად გვაქვს თუ რა ტიპისაა ცვლადი. მაგ.: **console.log(typeof x)-ის** შედეგი იქნება number (თუ არის 5, 7, 3...), ხოლო String(თუ არის ”zuzu” “57”).

```
/*****************************
* Basic operators
*/
/*
var year, yearJohn, yearMark;
now = 2018;
ageJohn = 28;
ageMark = 33;

// Math operators
yearJohn = now - ageJohn;
yeahMark = now - ageMark;

console.log(yearJohn);

console.log(now + 2);
console.log(now * 2);
console.log(now / 10);


// Logical operators
var johnOlder = ageJohn < ageMark;
console.log(johnOlder);


// typeof operator
console.log(typeof johnOlder);
console.log(typeof ageJohn);
console.log(typeof 'Mark is older tha John');
var x;
console.log(typeof x);
*/
```

2.6) ოპერაციების უპირატესობა (precedence)

ამ თავში ჩვენ განვიხილავთ ოპერაციების precedence-ს, სადაც ყოველ ოპერატორს თავისი იერარქიული შესრულების ნომერი აქვს. შეკრება, მეტობა, ტოლობა და ა. შ ყველა ეს ოპერატორი Asignment ოპერატორებს წარმოადგენენ. Asignment (მინიჭების) ოპერატორის რამდენიმე ვარიანტი გვაქვს - =, +=, -=, *=, /=, %=. ამ ყველა ოპერატორს აქვს თავისი

შესრულების მიმართულება (Associativily). მაგ.: x = x +1; x += 1 და x++ ყველა ამ ოპერატორს აქვს მიმართულება left-to-right.

```
JS app.js > ...
1   /*****************************
2   * Operator precedence
3   */
4
5   var now = 2018;
6   var yearJohn = 1989;
7   var fullAge = 18;
8
9   // Multiple operators
10  var isFullAge = now - yearJohn >= fullAge; // true
11  console.log(isFullAge);
12
13  // Grouping
14  var ageJohn = now - yearJohn;
15  var ageMark = 35;
16  var average = (ageJohn + ageMark) / 2;
17  console.log(average);
18
19  // Multiple assignments
20  var x, y;
21  x = y = (3 + 5) * 4 - 6; // 8 * 4 - 6 // 32 - 6 // 26
22  console.log(x, y);
23
24
25  // More operators
26  x *= 2;
27  console.log(x);
28  x += 10;
29  console.log(x);
30  x--;
31  console.log(x);
32
```

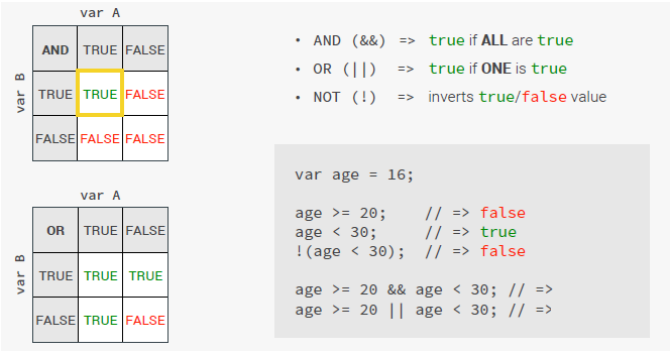
2.7) If-Else statement

როგორც სხვა პროგრამულ ენებში ასევე JS-შიც გვაქვს შესაძლებლობა გამოვიყენოთ სხვადასხვა კონტროლის საშუალება, რომელიც გვაძლევს უფლებას რაიმე კოდის ბლოკი ან შევასრულოთ ან არა, ასევე როდის შევასრულოთ ის, უფრო სწორად რა შემთხვევაში. **If/else statement-ზე** ვიყენებთ ფიგურულ ფრჩხილებს იმისათვის რომ გამოვყოთ პირობითი ბლოკი. პირობის შესადგენათ ჩვენ ვიყენებთ პირობით და ლოგიკურ ოპერატორებს. **დაიმახსოვრეთ ლოგიკურ ოპერატორებში = განსხვავდება = = =-საგან.**

```
JS app.js > ...
1   /*****************************
2   * If / else statements
3   */
4
5   var firstName = 'John';
6   var civilStatus = 'single';
7
8   if (civilStatus === 'married') {
9       console.log(firstName + ' is married!');
10  } else {
11      console.log(firstName + ' will hopefully marry soon □');
12  }
13
14
15  var isMarried = true;
16  if (isMarried) {
17      console.log(firstName + ' is married!');
18  } else {
19      console.log(firstName + ' will hopefully marry soon □');
20  }
21
```

2.8) Boolean logic (ლოგიკური კავშირები)

ამ თავში ჩვენ ვნახავთ Boolean logic-ის გამოყენებას if/else statement-ის შემთხვევაში. პირობით ოპერატორში ჩვენ გვაქვს მარტივი და რთული პირობა, რომლის შესრულებაც უწევს if statement-ს. მარტივი პირობის დროს ჩვეულებრივ გამოიყენებს პირობით ნიშნებს - <, >, ==, <=, >=, ხოლო რთული პირობის დროს გვჭირდება ლოგიკური AND(&&) და OR(||) კავშირები, რომლითაც ნებისმიერი ლოგიკური წყობის იმიტაცია შეგვიძლია შევასრულოთ. ჭეშმარიტების ცხრილიდან შეგვიძლია ვნახოთ რა შედეგი გვაქვს ლოგიკური კავშირის დროს:




```
JS app.js > ...
1   /*****************************
2   * Boolean logic
3   */
4
5   var firstName = 'John';
6   var age = 20;
7
8   if (age < 13) {
9       console.log(firstName + ' is a boy.');
10  } else if (age >= 13 && age < 20) {
11      console.log(firstName + ' is a teenager.');
12  } else if (age >= 20 && age < 30) {
13      console.log(firstName + ' is a young man.');
14  } else {
15      console.log(firstName + ' is a man.');
16  }
17
```

2.9) Ternary ოპერატორი (სამნავი)

პირობითი ოპერატორის გამარტივებული ვარიანტიც გვაქვს, რომელითაც ჩანაწერები ერთ ხაზზე ეტევა. ეს არის Ternary ოპერატორი სადაც გვაქვს:

<პირობა> ? კოდი ჭეშმარიტების დროს : კოდი მცდარობის შემთხვევაში.

სამნავ ოპერატორში ფიგურული ფრჩხილებით ბლოკის შემოტანა შეუძლებელია. მას (?_:_) სამნავი ქვია რადგან აქვს 3 operand (წევრები).

რაც შეეხება **Switch** ოპერატორს ისიც პირობითი ოპერატორის ერთერთი ტიპია, სადაც მის სინტაქსურ ჩანაწერში switch()-ში შემოგვაქვს ის ცვლადი რომლის შედარება გვინდა და ბლოკში ვწერთ "case"-ებს თუ რა შემთხვევაში რა გვინდა გაკეთდეს. ყოველი case-ის აღწერის ბოლოს არ დაგავიწყდეთ brake-ის მითითება, რაც switch-ის მუშაობას გარკვეული case-ის შესრულების შემდეგ შეწყვეტს.

app.js > ...

```
/*****************************
* The Ternary Operator and Switch Statements
*/
var firstName = 'John';
var age = 14;
// Ternary operator
age >= 18 ? console.log(firstName + ' drinks beer.') : console.log(firstName + ' drinks juice.');
var drink = age >= 18 ? 'beer' : 'juice';
console.log(drink);

(if (age >= 18) {
    var drink = 'beer';
} else {
    var drink = 'juice';
}
// Switch statement
var job = 'instructor';
switch (job) {
    case 'teacher':
    case 'instructor':
        console.log(firstName + ' teaches kids how to code.');
        break;
    case 'driver':
        console.log(firstName + ' drives an uber in Lisbon.');
        break;
    case 'designer':
        console.log(firstName + ' designs beautiful websites.');
        break;
    default:
        console.log(firstName + ' does something else.');
}

age = 56;
switch (true) {
    case age < 13:
        console.log(firstName + ' is a boy.');
        break;
    case age >= 13 && age < 20:
        console.log(firstName + ' is a teenager.');
        break;
    case age >= 20 && age < 30:
        console.log(firstName + ' is a young man.');
        break;
    default:
        console.log(firstName + ' is a man.');
}
```

2.10) Falsy and truthy value

ამ თავში ჩვენ გეტყვით რა არის falsy and truthy value, რაც აუცილებლად უნდა იცოდეს JS-ის დეველოპერმა.

Falsy value – undefined, null, 0, “”, Nan.

Truthy value - ყველა სხვა დანარჩენი.

app.js > ...

```
/*****************************
* Truthy and Falsy values and equality operators
*/

// falsy values: undefined, null, 0, '', NaN
// truthy values: NOT falsy values

var height;

height = 23;

if (height || height === 0) {
    console.log('Variable is defined');
} else {
    console.log('Variable has NOT been defined');
}

// Equality operators
if (height === '23') {
    console.log('The == operator does type coercion!');
}
```

2.11) რა არის Function და მისი მნიშვნელობა

ამ თავში აღვწერთ ერთ-ერთ მნიშვნელოვან დეტალს, რასაც ქვია function. ჩვენ პროგრამაში გვაქვს კოდის ისეთი ნაწილი, რომლის ხშირად გამოყენება გვიწევს. ამ შემთხვევაში ჩვენ შეგვიძლია ეს კოდი მოვათავსოთ ფუნქციაში და შემდეგ გამოვიძახოთ. ამ დროს ერთიდაიგივე კოდის გამეორებით წერა არ მოგვიწევს. ფუნქციას სჭირდება აღწერა, ანუ დეკლარაცია, სადაც გაწერილი იქნება მისი შესაძლებლობები. ამავდროულად შეგვიძლია მივცეთ არგუმენტები(პარამეტრები), რომელიც ამ პარამეტრების საშუალებით შეძლებს შედეგის დაბრუნებას. როდესაც ფუნქციის დეკლარირებას დავასრულებთ ამის შემდეგ პროგრამის ნებისმიერ line-ზე გამოვიძახებთ მას ფუნქციის სახელით და არსებული პარამეტრებით, რაც ან დაგვიბრუნებს შედეგს, ან სამუშაოს დამოუკიდებლად შეასრულებს

დასკვნა: ამ მეთოდის გამოყენებით ჩვენ გვაქვს პროგრამაში DRY (Don’t Repeat Yourself)-ის პრინციპი.

```
/*****************************
 * Functions
 */

function calculateAge(birthYear) {
    return 2018 - birthYear;
}

var ageJohn = calculateAge(1990);
var ageMike = calculateAge(1948);
var ageJane = calculateAge(1969);
console.log(ageJohn, ageMike, ageJane);


function yearsUntilRetirement(year, firstName) {
    var age = calculateAge(year);
    var retirement = 65 - age;

    if (retirement > 0) {
        console.log(firstName + ' retires in ' + retirement + ' years.');
    } else {
        console.log(firstName + ' is already retired.')
    }

}

yearsUntilRetirement(1990, 'John');
yearsUntilRetirement(1948, 'Mike');
yearsUntilRetirement(1969, 'Jane');
```

2.12) Function expression and Function statement

ზემოთ ჩვენ ვისწავლეთ function-ის გამოყენებბა, რომელსაც წინასწარ ვადეკლარირებთ. ამ მეთოდს ვეძახით function statement-ს, მაგრამ არის ფუნქციის გამოყენების expression მეთოდიც. ის შემდეგნაირად გამოიყურება

Var whatdoyoudo = Function(job, firstname) { }

მათ ძირეულ განსხვავებაზე chapter 3-ში ვისაუბრებთ. მანამდე კი უნდა დავიმახსოვროთ ის რომ statement method-ს არ შეუძლია დაგვიბრუნოს რაიმე, ხოლო expression method შედეგს გვიბრუნებს მითითებულ ცვლადში.

```
/*****************************
 * Function Statements and Expressions
 */

// Function declaration
// function whatDoYouDo(job, firstName) {}

// Function expression
var whatDoYouDo = function(job, firstName) {
    switch(job) {
        case 'teacher':
            return firstName + ' teaches kids how to code';
        case 'driver':
            return firstName + ' drives a cab in Lisbon.'
        case 'designer':
            return firstName + ' designs beautiful websites';
        default:
            return firstName + ' does something else';
    }
}

console.log(whatDoYouDo('teacher', 'John'));
console.log(whatDoYouDo('designer', 'Jane'));
console.log(whatDoYouDo('retired', 'Mark'));
```

2.13) Array in JS

ამ თავში ვისაუბრებთ array-ზე, მის initialization-ზე, method-ებზე, mutate-ზე. მასში შეგვიძლია სხვადასხვა ტიპის data მოვათავსოთ. მასზე მიმართვა შეგვიძლია პირდაპირ და ირიბად. მეთოდები რომლებიც მრავლად არის array-ში, მათი სიღრმისეული განხილვა შემდეგ თავებში იქნება. ახლა მინდა მისი პრაქტიკული გამოყენება შევძლოთ. .push(); .unshift(); .shift(); .pop(); .length(). მასივში ელემენტებს აქვს თავისი index-ის ნომერი და ათვლა ნულიდან იწყება.

app.js > ...

```
/*****************************
* Arrays
*/

// Initialize new array
var names = ['John', 'Mark', 'Jane'];
var years = new Array(1990, 1969, 1948);

console.log(names[2]);
console.log(names.length);

// Mutate array data
names[1] = 'Ben';
names[names.length] = 'Mary';
console.log(names);

// Different data types
var john = ['John', 'Smith', 1990, 'designer', false];

john.push('blue');
john.unshift('Mr.');
console.log(john);

john.pop();
john.pop();
john.shift();
console.log(john);

console.log(john.indexOf(23));

var isDesigner = john.indexOf('designer') === -1 ? 'John is NOT a designer' : 'John IS a designer';
console.log(isDesigner);
```

2.14) Object-ის მუშაობის პრინციპი JS

Object მნიშვნელოვანი მახასიათებელია JS-ში და მასში ვინახავთ value-ს, რომელზე პასუხისმგებლობას კონკრეტულ keyword-ს ვაძლევთ, შემდეგ რომ მარტივად მივწვდეთ. ზუსტად „ . “ გვჭირდება იმისათვის, რომ მივწვდეთ object-ის propert-ებს. ობიექტის შემოტანისას საკმარისია { } სადაც შემოგვაქვს keyword და მისი მნიშვნელობა რომელიც ნებისმიერი ტიპი შეიძლება იყოს მათ შორის მასივიც. Object-ის propert-ის რედაქტირებაც შეგვიძლია, ასევე შეგვიძლია მასში გვქონდეს მეთოდი და შეგვიძლია ახალი object-ის ინიციალიზაცია new object();

```
JS app.js > ...
1   /*****************************
2   * Objects and methods
3   */
4
5   var john = {
6       firstName: 'John',
7       lastName: 'Smith',
8       birthYear: 1992,
9       family: ['Jane', 'Mark', 'Bob', 'Emily'],
10      job: 'teacher',
11      isMarried: false,
12      calcAge: function() {
13          this.age = 2018 - this.birthYear;
14      }
15  };
16
17  john.calcAge();
18  console.log(john);
19
```

2.15) “this” pointer in object

ამ თავში ყურადღებას გავამახვილებთ this ოპერატორზე, რომელიც ობიექტის მიმთითებელია და მისი საშუალებით შეგვიძლია თავად object-ში მისივე property-ების გამოყენება. აქ გვექნება პრაქტიკული მაგალითები თუ როგორ შეიძლება object-ში ფუნქციების და მეთოდების შემოტანა. ასევე ვნახავთ თუ როგორ შეგვიძლია ახალი property-ების დამატება object-ში.

```
JS app.js > ...
1   ////////////////////////////////////
2   // Lecture: The this keyword
3
4
5   //console.log(this);
6
7   calculateAge(1985);
8
9   function calculateAge(year) {
10      console.log(2016 - year);
11      console.log(this);
12  }
13
14  var john = {
15      name: 'John',
16      yearOfBirth: 1990,
17      calculateAge: function() {
18          console.log(this);
19          console.log(2016 - this.yearOfBirth);
20
21          function innerFunction() {
22              console.log(this);
23          }
24          innerFunction();
25      }
26  }
27
28  john.calculateAge();
29
30  var mike = {
31      name: 'Mike',
32      yearOfBirth: 1984
33  };
34
35
36  mike.calculateAge = john.calculateAge;
37  mike.calculateAge();
38
```

2.16) Loop-ის განხილვა for, while

Loop გამოიყენება იმისათვის რომ კოდის ერთიდაიგივე line-ები არ განმეორდეს (ანუ ყველას ხელით წერა არ მოგვიწიოს). ამ შემთხვევაში ვიხილავთ for loop-ს. For-ის ჩაწერის რამდენიმე ვარიანტია, მას 3 პარამეტრი აქვს: **საწყისი ინდექსის ინიციალიზაციის ბლოკი, პირობის ლოგიკის ბლოკი (სადამდე გაგრძელდეს), ინდექსის ცვლილების ბლოკი.** ბოლოს for-ს აქვს თავისი { } შესრულების ბლოკი, რომელიც სრულდება ზედა სექციების ლოგიკის ხარჯზე. ამ შესრულების ბლოკში შეგვიძლია გამოვიყენოთ break და continue, რაც მაგალითში კარგად არის ახსნილი. ასევე for-ის ლოგიკას შეუძლია პროგრესი უკუ პროცესითაც წაიყვანოს.

JS app.js > ...

```
/*****************************
* Loops and iteration
*/
// for loop
for (var i = 1; i <= 20; i += 2) {
    console.log(i);
}

// i = 0, 0 < 10 true, log i to console, i++
// i = 1, 1 < 10 true, log i to the console, i++
//...
// i = 9, 9 < 10 true, log i to the console, i++
// i = 10, 10 < 10 FALSE, exit the loop!


var john = ['John', 'Smith', 1990, 'designer', false, 'blue'];
for (var i = 0; i < john.length; i++) {
    console.log(john[i]);
}

// While loop
var i = 0;
while(i < john.length) {
    console.log(john[i]);
    i++;
}


// continue and break statements
var john = ['John', 'Smith', 1990, 'designer', false, 'blue'];

for (var i = 0; i < john.length; i++) {
    if (typeof john[i] !== 'string') continue;
    console.log(john[i]);
}

for (var i = 0; i < john.length; i++) {
    if (typeof john[i] !== 'string') break;
    console.log(john[i]);
}

// Looping backwards
for (var i = john.length - 1; i >= 0; i--) {
    console.log(john[i]);
}

```

2.17) ES5, ES6 – ES2015, ES6 +

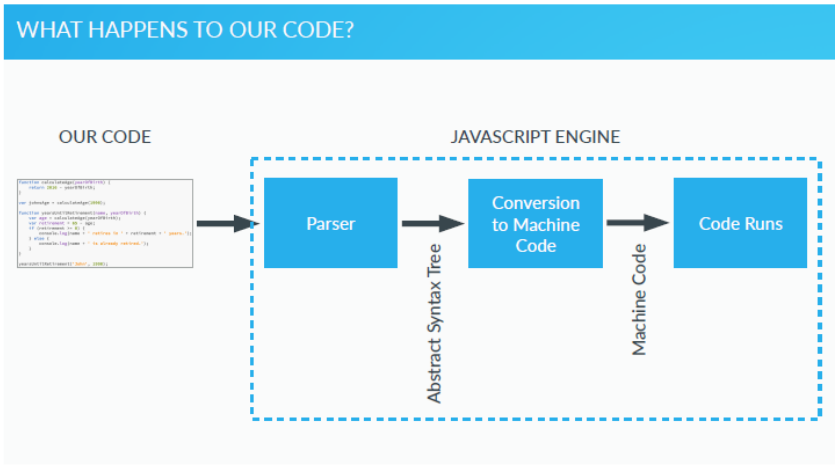
ამ ეტაპის ბოლო თავში საუბარი გვექნება JS-ის ისტორიაზე. იმას რასაც ახლა გეტყვით JS-ის ყველა დეველოპერმა უნდა იცოდეს, თუ საიდან მოდის JS. ის პირველად 1995 წელს გამოჩნდა “livescript”-ის სახელით, რაც მალევე შეიცვალა javascript-ით. ეს მარკეტინგული გათვლა იყო, რადგან მაშინაც java დიდი პოპულარობით სარგებლობდა. ასე რომ ამ ორ ენას

ერთმანეთთან არანაირი კავშირი არ აქვთ. Javascript მალე გახდა esma კორპორაციის ნაწილი და შეიქმნა ახალი სტანდარტი esmascript_ES1. ჩვენ განვიხილავთ ES5-ს რომელიც 2009 წლიდან არის და 2015 უკვე ცვლილებებით მოგვევლინა ES6(ES2015)-ის სახელით. აქედან მოყოლებული ყოველ წელს ES-ის ახალი ვერსია გამოდის, მაგრამ ES6-ისაგან ძირეული განსხვავებები არ არის. ამ სიახლეს ES next-ს ეძახიან ან ES7, ES8, ES9 და ა.შ. ES5-ის მხარდაჭერა ყველა browser-ს აქვს, ხოლო ES6-ის და ახალი თაობების მხარდაჭერა მხოლოდ ახალ browser-ებს აქვთ. ეს ცხრილი შეგვიძლია ინტერნეტში ვნახოთ. ამავდროულად არსებობს tools-ები რომლებიც მოდერნიზებულ JS კოდს უკან ES5-ში გადმოიყვანს. ამ ეტაპზე ჩვენ ვირჩევთ ES6-ის განხილვას და მის გამოყენებას, რადგან მან უკვე კარგად მოიკიდა ფეხი, მაგრამ საჭიროდ ჩავთვალე ES5-ით დამეწყო რადგან ჯერჯერობით solution-ები ძველი სინტაქსითაა და community ჯერ კიდევ მრავლად ყავს.

Chapter 3

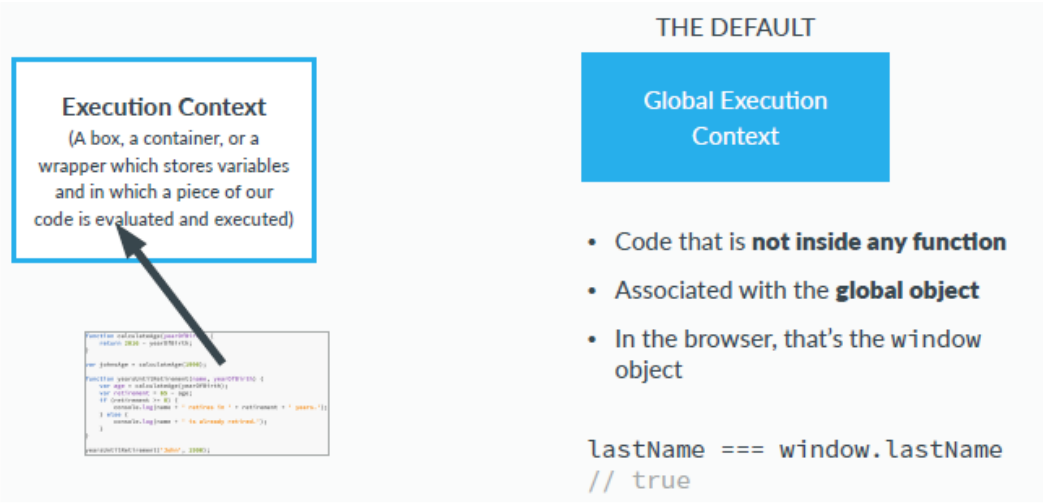
3.1) How our codes are executed: JS parses and engines

ამ თავში თეორიულად განვიხილავთ თუ რა ხდება სცენის უკან, როდესაც JS კოდი ეშვება. ძირითადად JS კოდს სადაც ვუშვებთ ეს არის გარემო chrome, Opera, Mozilla და ა.შ არის ასევე სხვა გარემო (environment), სადაც JS-ის გაშვება შეგვიძლია, მაგალითად node JS რომელიც web-server არის. Browser-ში არის Javascript engine რომელიც არის JS კოდზე პასუხისმგებელი. პირველ რიგში ისინი parsing-ს (კოდის parse) აკეთებენ, რითაც ამოწმებენ სინტაქსურ შეცდომებს, ესე იგი მან ზედმიწევნით იცის javascript-ის წესები. მსგავსი ძრავებია google V8 engine, Spider monkey, Javascript core და ბევრი სხვა მრავალი. გამოდის chrome-ს თავისი engine აქვს. Spider monkey არის ერთ-ერთი ძველი ძრავი, რომლის მხარდაჭერა Netscape-ს ქონდა თავის დროზე. Parser თუ შეცდომას დააფიქსირებს, შეწყვეტს execute-ს და გამოიტანს error-ს. თუ წარმატებით გაივლის ეს ფაზა მაშინ შექმნის data სტრუქტურას სახელად Abstract Syntax Tree რომელიც გადავა მანქანურ კოდში

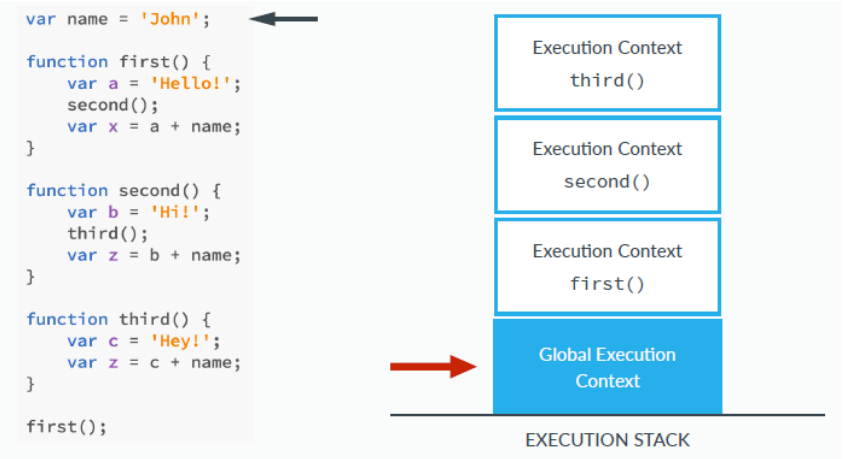


3.2) Execution context and Execution stack

წინა ლექციაში ჩვენ განვიხილეთ Javascript-ის engine-ს მუშაობის კონცეფცია და ვნახეთ რომ parsing-ის მერე კოდს მანქანურ ენაში გადაიყვანს. ახლა მე ფოკუსირებას გავაკეთებ იმაზე თუ როგორ ხდება ამ კოდის შესრულება ანუ execution context. ყველა ის გარემო სადაც უნდა შესრულდეს JS კოდი არის execution context და ის არის კონტეინერში სადაც ინახავს კოდს, რომლის შეფასება და გაშვება (execute) ხდება, ის Global Execution სახელით არის ცნობილი. სახელიც მიგვითითებს მთავარ context-ზე, რომელიც გლობალურად ინახავს ყველა ცვლადს და ფუნქციას. მაგრამ მკაცრად დაიმახსოვრეთ **function-ის შიგნით არსებული კოდი არ შედის!.**



ანუ შეგვიძლია წარმოვიდგინოთ, რომ execution context არის object-ივით. Global execution context ასოცირდება global object-თან ანუ window-სთან, რომელიც browser-ს გააჩნია. მაგ.: x ტოლფასია window.x-ის, ერთი და იგივეა და ბოლოს როდესაც ფუნქციის გამოძახება ხდება, იქმნება new execution context. თუ რამდენიმე ფუნქციის გამოძახება გვინდა, შეიქმნება execution stack რომლის შესრულება მოხდება იერარქიულად.მაგალითში ძალიან კარგად ჩანს ეს თემა:



ამ მაგალითში პირველი არის ცვლადის დეკლარაცია, რომელიც global context-შია და ამასთანავე ყველა ფუნქციის დეკლარაცია global context-ში ინახება. ბოლოს ხდება პირველი ფუნქციის გამოძახება რომელიც შექმნის თავის first execution context. აქ თავის მხრივ გვაქვს ცვლადი რომლის დეკლარირება ხდება, რომელიც უკვე ამ local execution context-ში არის, რაც იმას ნიშნავს რომ მისი დანახვა Global Execution-იდან ვერ მოხდება. იგივე მოხდება მე-2 მე-3 ფუნქციის გამოძახებისას რომლებიც შექმნიან თავიანთ execution context-ებს. მათ აქვთ თავიანთი დეკლარირებული ცვლადები. საბოლოოდ მივიღეთ execution stack-ს რომლის შესრულება მოხდება შესაბამისი რიგითობით. ამ stack-ში როდესაც მესამე ფუნქცია დაასრულებს სამუშაოს გადავა მეორეზე, შემდეგ პირველზე და ბოლოს თუ global context-ში დარჩება რაიმე გასაკეთებელი დაასრულებს მას. შემდეგ ლექციაში დეტალურად ვნახავთ როგორ იქმნება execution context და რატომ არის ის ასე მნიშვნელოვანი.

3.3) Execution context in detail: creation and execution phases, hoisting

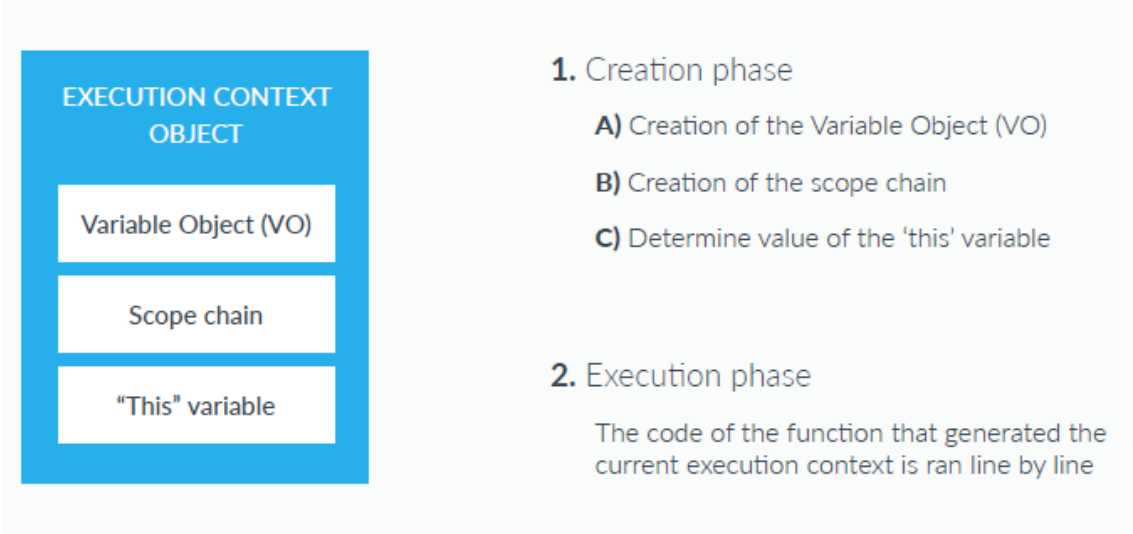
წინა მაგალითში ვნახეთ თუ როგორ შეიქმნა execution stack, მაგრამ ეხლა უნდა ვნახოთ კონკრეტულად თუ რა ფაზებს გადის, როდესაც იქმნება execution context. ჩვენ ის შევადარეთ object-ს, რომელშიც თუ ჩავუღრმავდებით, ინახება **variable object; scope chain და this.** როდესაც function-ს ვიძახებთ, სანამ ის execution stack-ში მოხვდება, მანამდე ხდება ორი ფაზა:

Creation phase

- Creation of the variable object (VO)
- Creation of the scope chain
- Determining value of the “this” variable

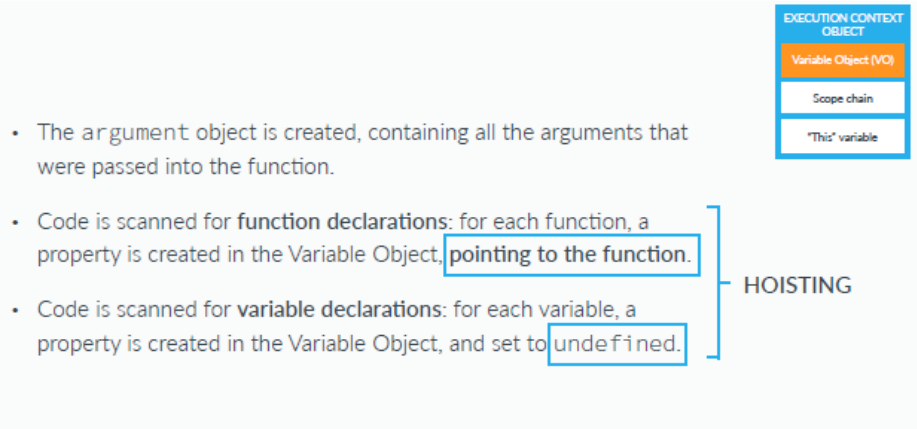
Execution phase

- იქმნება context რომელშიც დაგენერირებულია კოდი და სრულდება line-to-line



იმ არგუმენტებს რასაც function-ს მივცემთ პარამეტრი/property-ს სახით, ან მის context-ში არის დეკლარირებული, აისახება variable object-ში. ასევე ჩანაწერში დეკლარირებული ფუნქციებიც აისახება VO-ში. ამას ეძახიან hoisting რასაც შემდეგ თავში დეტალურად განვიხილავთ.

დასკვნა: ყველა execution object-ს აქვს თავისი Object, რომელიც ბევრ მნიშვნელოვან მონაცემს ინახავს და იყენებს მათ სანამ ფუნქცია გაეშვება.



3.4) Hoisting in practice

აქ ავხსნით hoisting-ის თემას და დავიმახსოვროთ, რომ ფუნქციის შემთხვევაში hoisting მუშაობს მხოლოდ function declaration-ის დროს. ხოლო function expression-ის დროს არ მუშაობს, error-ს მოგვცემს. ცვლადების შემთხვევაში კი hoisting მუშაობს

```
JS app.js > ...
1   ////////////////////////////////////
2   // Lecture: Hoisting
3
4
5   // functions
6   calculateAge(1965);
7
8   function calculateAge(year) {
9       console.log(2016 - year);
10  }
11
12  // retirement(1956);
13  var retirement = function(year) {
14      console.log(65 - (2016 - year));
15  }
16
17
18  // variables
19
20  console.log(age);
21  var age = 23;
22
23  function foo() {
24      console.log(age);
25      var age = 65;
26      console.log(age);
27  }
28  foo();
29  console.log(age);
30
```

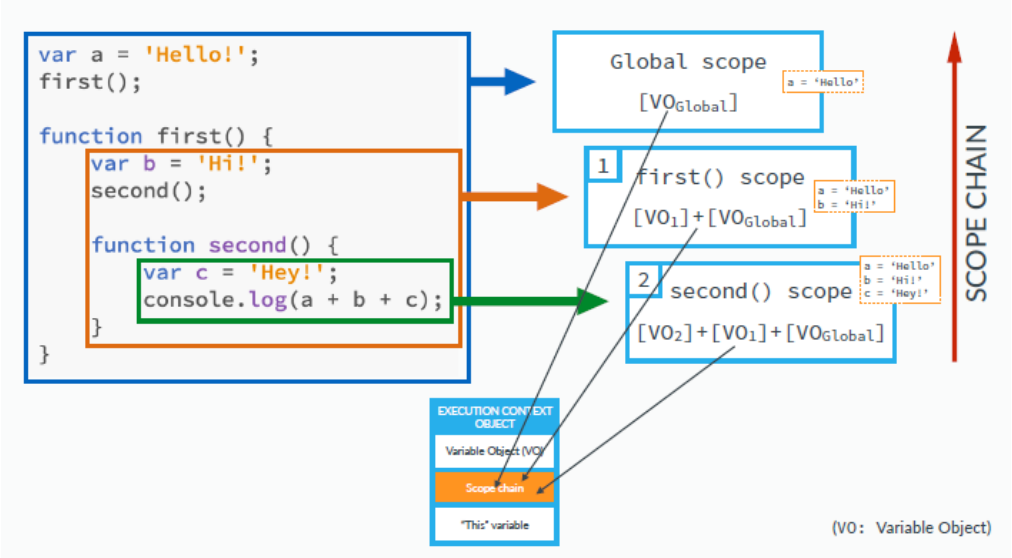
აქ მაგალითში კარგად ჩანს რომ ფუნქციაში დეკლარირებული ცვლადი რომლის სახელია x, განსხვავდება global-ში არსებული x-ისაგან მიუხედავად იმისა რომ სახელი ერთიდაიგივე აქვთ. ეს ხდება იმიტომ რომ პირველ შემთხვევაში ის ფუნქციის execution context-ში ინახება, ხოლო მეორეში global execution context-ში. დაიმახსოვრეთ რომ hoisting ფუნქციის შიგნითაც მუშაობს.

3.5) Scope and scope chain

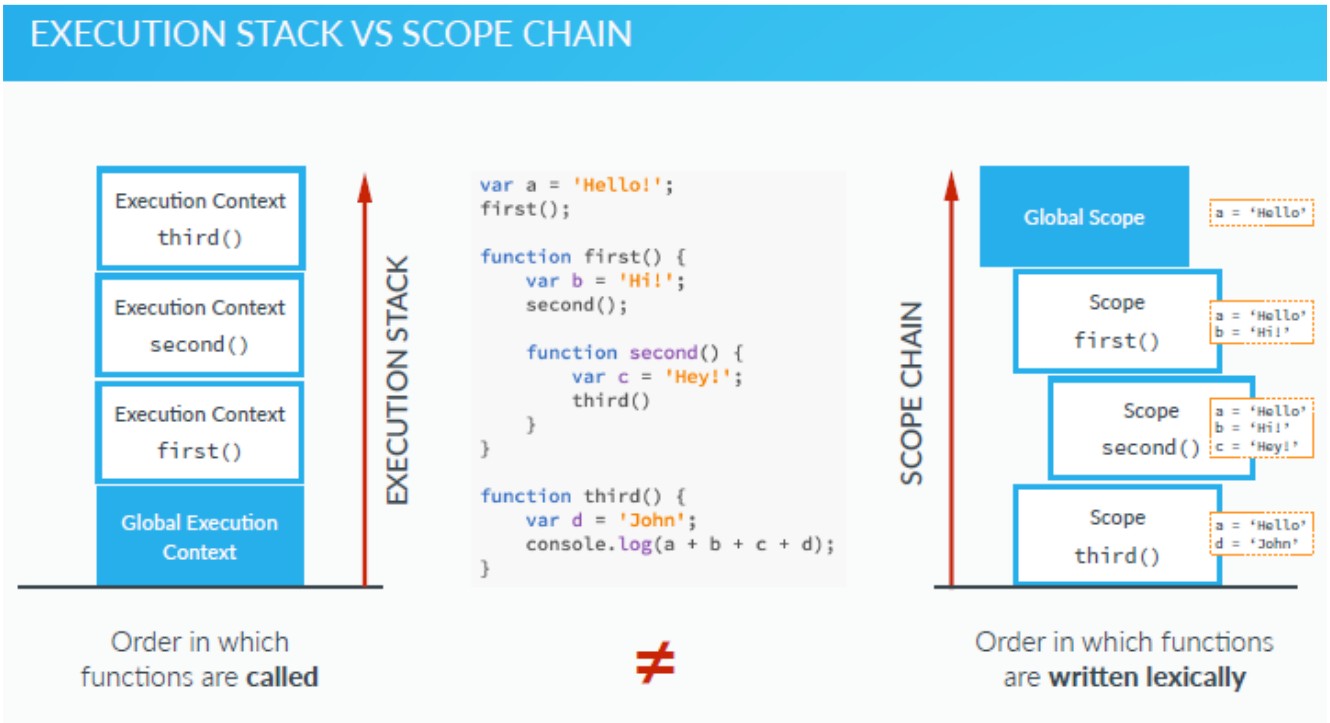
ეს არის ლოგიკა, თუ რა დროს არის ცვლადი ან ფუნქცია ხელმისაწვდომი. Global scope არის იგივე default scope, რომელზეც წვდომა ყველას შეუძლია. აქ იგულისხმება ის რომ მის object-ში არსებულ property-ებს შეხედავს ყველა.

```
JS app.js > ...
1   ////////////////////////////////////
2   // Lecture: Scoping
3
4
5   // First scoping example
6   var a = 'Hello!';
7   first();
8
9   function first() {
10      var b = 'Hi!';
11      second();
12
13      function second() {
14          var c = 'Hey!';
15          console.log(a + b + c);
16      }
17  }
18
19
20  // Example to show the differece between execution stack and scope chain
21  var a = 'Hello!';
22  first();
23
24  function first() {
25      var b = 'Hi!';
26      second();
27
28      function second() {
29          var c = 'Hey!';
30          third()
31      }
32  }
33
34  function third() {
35      var d = 'John';
36      //console.log(c);
37      console.log(a+d);
38  }
39
```

მაგალითი ძალიან კარგად გვიჩვენებს ამ დამოკიდებულებას თუ როგორ იქმნება scope chain. Lexical scope არის როდესაც ჩადგმული არის context-ები და მშობლის პრინციპით მის ცვლადებსაც ვხედავთ.



მაგრამ სხვაგვარად არის საქმე როდესაც context-ებს საერთოდ წინაპარი არ ყავთ global context-მდე. ამ შემთხვევაში ისინი ერთმანეთის ცვლადებს ვერ შეხედავენ, რაც მაგალითში კარგად ჩანს. Scope - ეს ლოგიკა სხვა პროგრამულ ენებს if/for ბლოკებზეც აქვთ მაგრამ Javascript-ში სხვაგვარდა არის. Execution context stack და scope chain ერთმანეთისგან განსხვავდებიან და ეს მაგალითში კარგად ჩანს.



3.6) “This” keyword

ჩვენ წინა თავებში განვიხილეთ execution context და ვიცით რომ ყველას აქვს object-ები. This ზუსტად ინახავს ამ object-ებს. Default-ად this ინახავს global context object-ს window-ს,

რომელიც browser-ის შემთხვევაში default object-ია. ასევე როდესაც მეთოდში გამოვიყენებთ this-ს ისიც global object-ზე მიგვითითებს, მაგრამ როდესაც ჩვენ object-ს შემოვიტანთ, იქ გამოყენებული this მიმთითებელი მიმდინარე ობიექტს გვიჩვენებს.თუ ჩვენ ამ ობიექტში/მეთოდში კიდევ ობიექტს შემოვიტანთ, სადაც გამოვიყენებთ ამ მიმთითებელს მაშინ ის გვიჩვენებს global object-ს (window-ს). ამას JS-ის community ზოგი ეთანხმება ზოგი არა. მაგალითში კარგად ავხსენით თუ როგორ ხდება method borrowing სადაც ერთი ობიექტიდან მეთოდს ვანიჭებთ მეორე ობიექტის მეთოდს.

app.js > ...

```
1  ////////////////////////////////////
2  // Lecture: The this keyword
3
4
5  //console.log(this);
6
7  calculateAge(1985);
8
9  function calculateAge(year) {
10     console.log(2016 - year);
11     console.log(this);
12 }
13
14 var john = {
15     name: 'John',
16     yearOfBirth: 1990,
17     calculateAge: function() {
18         console.log(this);
19         console.log(2016 - this.yearOfBirth);
20
21         function innerFunction() {
22             console.log(this);
23         }
24         innerFunction();
25     }
26 }
27
28 john.calculateAge();
29
30 var mike = {
31     name: 'Mike',
32     yearOfBirth: 1984
33 };
34
35
36 mike.calculateAge = john.calculateAge;
37 mike.calculateAge();
38
```

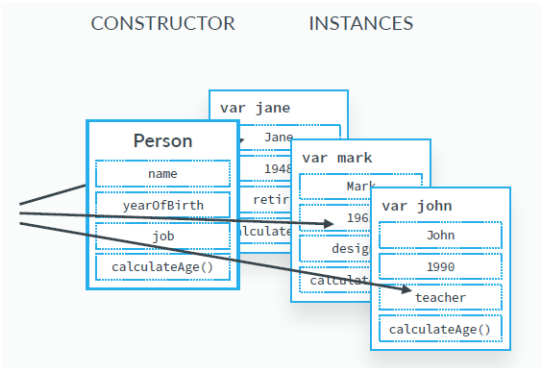
Chapter 5 Advanced Javascript: objects and function

5.1) ყველაფერი ობიექტის შესახებ inheritance and the prototype chain

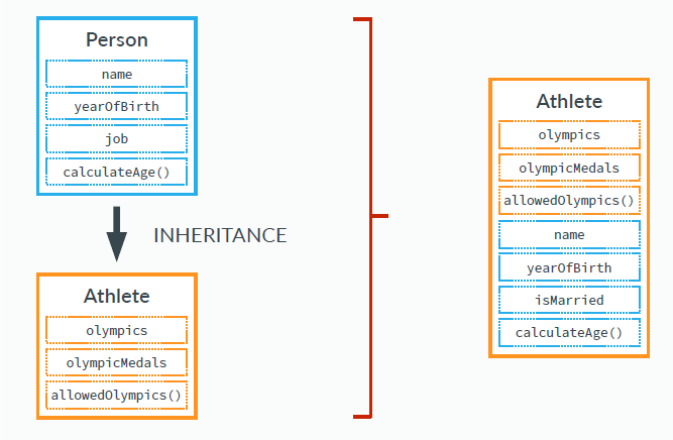
ამ თავში ჩვენ დაწვრილებით განვიხილავთ Object-ს, ვისაუბრებთ ობიექტზე ორიენტირებულ პროგრამირებაზე ზოგადად, მემკვიდრეობაზე თუ როგორ არის ობიექტში ის ხელმისაწვდომი javascript-ში prototype-ის საშუალებით. ბევრი ამბობს javascript-ში ყველაფერი არის object რაც 100%-ით სიმართლეს არ შეესაბამება რადგან JS-ში ჩვენ გვაქვს 2 ძირითადი ტიპის მნიშვნელობა primitives and objects. პრიმიტიული ტიპებია: numbers, strings, Boolean, undefined და ყველა სხვა დანარჩენი არის object.

ამიტომაც ამბობენ რომ ყველაფერი javascript-ში არის object. მართლაც array, function ან data არის ობიექტი js-ში რადგან javascript- ისთვის object არ არის მხოლოდ მონაცემების შემნახველი. ჩვენ object-ს ვიყენებდით მაგრამ სიღრმისეულად მას არ ვიცნობდით და ზუსტად object-ებია რომელიც javascript-ს განასხვავებს ყველა სხვა პროგრამული ენებისაგან. JS-ში მართლაც ყველაფერი object-ია.

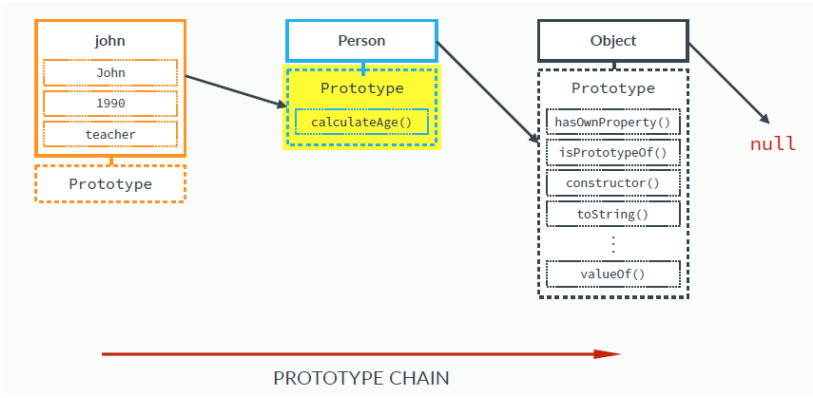
Object oriented პროგრამირება იყენებს object propert-ებს და მეთოდებს, ასევე შესაძლებელია ერთი ობიექტის მეორესთან ურთიერთქმედება იმისათვის რომ შევქმნათ უფრო რთული პროგრამული ლოგიკა. ამ სტრუქტურით ჩვენი კოდი იქნება სუფთა და ამოცანის ლოგიკასთან ახლოს. მაგალითად ჩვენ გაგვიკეთებია ობიექტი john რომელშიც გვქონდა propert-ები და რამდენი პიროვნებაც გვინდოდა შემოგვქონდა ახალ ახალი ობიექტები, მაგრამ არის უფრო სწორი გზა რომელსაც სხვა პროგრამულ ენებში class ეძახიან ჩვენ კი მასს კონსტრუქტორს ან prototype დავუძახებთ.



მაგ.: Person prototype რომლითაც ჩვენ თავის მხრივ შევქმნით ბევრ პიროვნებას ანუ მის მემკვიდრეებს რომელთა propert-ები უკვე განსაზღვრულია და მაგალითში ჩანს john, mark object-ები. მათ აქვთ თავიანთი propertები თავისი მნიშვნელობებით და ისინი ამავდროულად person-ში არსებულ მეთოდებსაც გამოიყენებენ, რადგან ისინი person-ის მემკვიდრეები არიან. ამ ყველაფერს პრაქტიკულად შემდეგ ლექციაში ვნახავთ.



“inheritance” არის როდესაც ერთი ობიექტი იზიარებს მეორე ობიექტის propert-ებს. მაგ.: „ათლეტი„ ისევ person-ის propert-ებს იზიარებს რადგან მასაც აქვს სახელი, ასაკი და შეგვიძლია ისინი გავაერთიანოთ. ეს ძალიან პრაქტიკული დეტალი არის რომელიც ამოცანის სწორად ინტერპრეტაციაში მოგვეხმარება, ნაკლები კოდის წერა მოგვიწევს, ჩვენი კოდი იქნება უფრო დინამიური და ლოგიკური. ეხლა ძალიან მნიშვნელოვანია თეორიულად გავიგოთ თუ როგორ მუშაობს inheritance. Javascript არის prototype base language, რაც იმას ნიშნავს რომ inheritance მუშაობს prototype-ს ხარჯზე და პრაქტიკულად ყველა javascript-ის object-ს აქვს prototype-ს property რომელიც inheritance-ის საშუალებას გვაძლევს JS-ში.



თეორიულად თუ გვინდა რომ john-ს ქონდეს მეთოდი მაგ.: calcAge ამისათვის კონსტრუქტორის person-ის prototype-ში უნდა ჩავამატოთ ეს მეთოდი. პრაქტიკულად ეს მეთოდი გვექნება person-ში მაგრამ inheritance-ს საშუალებით john მიწვდება ამ მეთოდს რომელიც მის prototype-ში იქნება და ყველა ის object რომელიც person კონსტრუქტორით შეიქმნება, ყველა გამოიყენებს calcAge მეთოდს. ასევე ძალიან მნიშვნელოვანია ვიცოდეთ

რომ თავად person არის სხვისი მემკვიდრე რომელიც არის object თავის მხრივ. ამ ლოგიკით ჩვენ გვაქვს prototype chain რა ლოგიკითაც როდესაც ობიექტში გამოვიძახებთ მეთოდს ის პრიველ რიგში მოძებნის თავის თავში, თუ ვერ იპოვა გადავა object-ის prototype-ში რომელიც არის მშობელი ობიექტის property. საბოლოოდ თუ ვერ იპოვა შედეგი იქნება null, რაც იმას ნიშნავს რომ ძებნა ფსკერამდე დავიდა და undefined შედეგს დააბრუნებს.

5.2) Creating objects: Function constructors

ამ თავში პრაქტიკულად შევქმნით ობიექტს Function constructor-ის საშუალებით. მუშაობისას ჩვენ ვიცით როგორ უნდა შევქმნათ ობიექტი “object literal”-ის ({ }) საშუალებით, მაგალითად john, მაგრამ ეხლა წარმოიდგინეთ ჩვენ გვჭირდება ყველა ჩემი მოსწავლის ობიექტების შექმნა თავისი სახელით, ასაკით და ჰობით. იმ გზით რაც ჩვენ ვიცით არა პრაქტიკული მეთოდია, მაგრამ ახლა ვისწავლით უფრო ხელსაყრელ მეთოდს. ეს “pattern” არის მიღებული მეთოდი რომელსაც function constructor-ს ვეძახით.

```
// Lecture: Function constructor
var john = {
    name: 'John',
    yearOfBirth: 1990,
    job: 'teacher'
};

var Person = function(name, yearOfBirth, job) {
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}

Person.prototype.calculateAge  = function() {
    console.log(2016 - this.yearOfBirth);
};

Person.prototype.lastName = 'Smith';

var john = new Person('John', 1990, 'teacher');
var jane = new Person('Jane', 1969, 'designer');
var mark = new Person('Mark', 1948, 'retired');

john.calculateAge();
jane.calculateAge();
mark.calculateAge();

console.log(john.lastName);
console.log(jane.lastName);
console.log(mark.lastName);
```

პირველ რიგში გავაკეთებთ person-ის ობიექტის დეკლარირებას Function constructor-ის მეთოდით, სადაც არგუმენტის სახით გვექნება name; age; job რომელთაც ობიექტის propert-ებში გადავიყვან this-ის გამოყენებით. This.name=name, და ა.შ. ამის შემდეგ შევქმნით ახალ object-ს john-ს new ოპერატორით სადაც არგუმენტების სახით შევიყვან მონაცემებს და ამ ხერხით ობიექტის შექმნას ვეძახით instanitation რადგან ეს ობიექტი არის person ობიექტის instance. ეხლა დეტალურად ავხსნით თუ რა არის new. ის ქმნის empty object-ს, ამის შემდეგ ვიძახებთ ფუნქციას person-ის სახელით და თავისი არგუმენტებით. ეს ფუნქცია ქმნის execution context სადა არის this variable (გახსოვთ) რომელიც default-ად global object-ის

მიმთითებელია, მაგრამ გაიხსენეთ ჩვენ new ოპერატორი გამოვიყენეთ რადგან ის ქმნის ახალ ცარიელ object-ს და ეს this-იც იქნება ამ ახლად შექმნილი ობიექტის მიმთითებელი.

ეხლა ვნახოთ პრაქტიკულად როგორ მუშაობს inheritance prototype-ს დახმარებით. ამისათვის კონსტრუქტორში დავამატოთ მეთოდი calcAge და this-ის გამოყენებით გამოვიყენოთ ამავე ობიექტის მახასიათებელი კერძოდ „yearOfBirth“. რის შემდეგაც ეს მეთოდი აღმოჩნდება john-ის object-ში და ის შეძლებს ამ მეთოდის გამოყენებას და თუ ჩვენ შევქმნით რამოდენიმე პიროვნებას ყველა მათგანი გაიზიარებს ამ propert-ებს. ჩვენ თეორიულად ვიცით რომ თუ constructor-ში დავამატებთ method-ებს inheritance-ის საშუალებით ის ხელმისაწვდომი იქნება ყველასათვის. Person prototype აქ თუ person-ის prototype-ში დავამატებთ propert-ს ეს მემკვიდრეობით გადაეცემა ყველას.

5.3) The prototype chain მაგალითები array-ს შემთხვევაში

ამ თავში კიდევ უფრო სიღრმისეულად ჩავიხედოთ Object-ში და მათ prototype-ის propert-ში console გვაძლევს საშუალებას რომ ეს ყველაფერი ვნახოთ და წინა მაგალითიდან მოდი console-ში ვნახოთ john და რას გვიჩვენებს chrome-ს console. ვხედავთ john-ის propert-ებს და ცვლადს _proto_ რომელიც ამ object-ის prototype-ია თავის მხრივ მშობლის propert-ები ანუ person-ის. თუ ცალკე ვნახავთ person-ს console-ში ვნახავთ რომ მისი propert-ები იგივეა რაც john-ის _proto_-ში არსებული propert-ები და თავის მხრივ person-ს აქვს თავისი _proto_ რომელიც მიუთითებს global object-ის propert-ებზე. პატარა მაგალითს გაჩვენებთ ამის დასამტკიცებლად.

John._proto_ = = = Person.prototye => ამის შედეგი იქნება True და ესეც უნდა იყოს.

ეხლა მინდა მოგიყვანოთ prototype chain-ის მაგალითი სადაც John-ის Age რომლის ტიპი არის number(typeof-ით შეგვიძლია შევამოწმოთ) მინდა რომ დავაკონვერტირო string-ში, მაგრამ მსგავსი მეთოდი ჩვენ არ გვაქვს? მოდი ვცადოთ console.log(typeof john.age.tostring()); ვნახეთ რომ შევძელით ეს მაგრამ როგორ? Tostring() მეთოდი იმალება object-ის prototype-ში. ჯერ მოიძიებდა john-თან შემდეგ person-ის propert-ებში და ბოლოს object-ის propert-ებში. ეს არის ზუსტად prototype chain-ის მუშაობა. ასევე შეგვიძლია გამოვიყენოთ სხვა მეთოდიც john.hasownproperty('calcAge') => false მაგრამ john.hasownproperty('age') => true, ასევე შეგვიძლია ვნახოთ ესეთი რამ john instanceof object => true. მოდი ეხლა ვნახოთ console-ში

```
var x = [2,7,9]
x
console.info(x)
```

სადაც კარგად ჩანს ის ინფორმაცია რომელიც კულისების მიღმაა და ჩვენ უკვე გვესმის საიდან მოდის ეს მონაცემები. აქვე არის array-ს prototype pop; shift და ა.შ.

5.4) Creating objects: Object.create

ობიექტის შექმნა გარდა inherit object prototype-ისა შესაძლებელია Object.create მეთოდითაც. ამ შემთხვევაში ჩვენ ჯერ ვაკეთებთ prototype-ს object-ს { } რომელსაც დავარქმევ personProto-ს დიდი „P“ ასოთი არ დავიწყებ რადგან ეს არ არის “function constructor”-ი. ამ ობიექტში შევიტანთ მეთოდს calcAge-ის სახელით და შემდგომ შევქმნით var john = object.create(personProto); რომელსაც პარამეტრის სახით გავატანეთ მეთოდი რომელსაც prototype-ში ჩასვამს. John ობიექტში propert-ების განსაზღვრას ჩვეულებრივი წესით შევძლებთ john.name = ‘john’; john.age = ‘30’; და ა.შ. ახლა შევქმნათ მეორე პიროვნება Tsotne. რომელსაც ობიექტის შექმნისას prototype არგუმენტის გარდა გავატანოთ მეორე არგუმენტი, რომელსაც შემდეგი სახე უნდა ქონდეს { name: { value: Tsotne } } უცნაურად ჩანს მაგრამ ესეთი სინტაქსი აქვს. მაგალითში კარგად ჩანს ყველა ეს დეტალი. მთავარი განსხვავება object-ის შექმნის create მეთოდისა და function constructor-ის არის ის რომ create-ის დროს უფრო მარტივად შეგვიძლია რთული მემკვიდრეობის აღწერა.

app.js > ...

```
////////////////////////////
// Lecture: Object.create

var personProto = {
    calculateAge: function() {
        console.log(2016 - this.yearOfBirth);
    },
    Lname: 'qartveli'
};

var tsotne = Object.create(personProto);
tsotne.name = 'tsotne';
tsotne.yearOfBirth = 1989;
tsotne.job = 'teacher';

var gio = Object.create(personProto, {
    name: { value: 'gio' },
    yearOfBirth: { value: 1969 },
    job: { value: 'designer' }
});

```

5.5) Primitives vs Objects

ამ თავში გამახვილებ ყურადღებას თუ რა განსხვავება არის primitive-სა და object-ს შორის. ცვლადი რომელიც წარმოადგენს პრიმიტიული ტიპის ცვლადს. ის თავის მხრივ ინახავს მონაცემებს, ხოლო ობიექტის შემთხვევაში ცვლადი რომელიც წარმოადგენს object-ს. ის არის მხოლოდ მიმთითებელი ოპერატოულ მეხსიერებაში, სადაც რეალურად არის object განთავსებული. მაგალითში ძალიან კარგად ჩანს რომ პრიმიტიული ცვლადის შემთხვევაში როდესაც a-ს ცვლილება გავაკეთე b-ს მნიშვნელობაში ცვლილება არ მომხდარა, რადგანაც მათ დამოუკიდებელი კოპირებული data აქვთ და არანაირ კავშირში არ არიან ერთმანეთთან. Object-ის შემთხვევაში კი პირიქით. ძალიან კარგად ჩანს მაგალითში რომ obj1 რომელიც შევქმენით თავისი propert-ებით და ეს obj1 მივაკუთვნეთ obj2-ს, როდესაც obj2-ში გავაკეთებ

ცვლილებას რომელიმე propert-ზე ამას obj1-იც დაინახავს, რადგან რეალურად ისინი ერთ object-ს ხედავენ და ისინი არიან უბრალოდ მიმთითებლები. კარგი მაგალითია როდესაც ფუნქციას არგუმენტის სახით უგზავნი პრიმიტიულ ცვლადს და object-ს ფუნქციის შიგნით პრიმიტიული მონაცემის კოპი გააკეთა a-ში ხოლო b-ში მან object-ზე მიმთითებელი შექმნა რომელიც იგივე object-ზე გააკეთებს მანიპულირებას, რასაც console.log-ით კარგად დავინახავთ.

```
JS app.js > ...
1   ////////////////////////////
2   // Lecture: Primitives vs objects
3
4   // Primitives
5   var a = 23;
6   var b = a;
7   a = 46;
8   console.log(a);
9   console.log(b);
10
11
12
13  // Objects
14  var obj1 = {
15      name: 'tsotne',
16      age: 26
17  };
18  var obj2 = obj1;
19  obj1.age = 30;
20  console.log(obj1.age);
21  console.log(obj2.age);
22
23  // Functions
24  var age = 27;
25  var obj = {
26      name: 'gio',
27      city: 'kutaisi'
28  };
29
30  function change(a, b) {
31      a = 30;
32      b.city = 'tbilisi';
33  }
34
35  change(age, obj);
36
37  console.log(age);
38  console.log(obj.city);
39
```

5.6) First class function: passing functions as arguments

ჩვენ ვიცით რომ ფუნქცია მსგავსია ობიექტის და შესაბამისად მას ობიექტის მსგავსად აქვს მსგავსი მახასიათებლები: მაგალითად ის არის, იქცევა ობიექტის მსგავსად. შეგვიძლია ფუნქცია მოვათავსოთ ცვლადში, შეგვიძლია ფუნქციას არგუმენტის სახით მივცეთ თავად ფუნქცია, ან კიდევ ფუნქციას შეგვიძლია დავაბრუნებინოთ თავად ფუნქცია შედეგად. ყველა ამ თვისებით ძალიან ბევრ შესაძლებლობებს იძლევა JS. ამ ლექციაში მე პრაქტიკულად გაჩვენებთ თუ როგორ შეიძლება ფუნქციას არგუმენტის სახით გავუგზავნოთ თავად ფუნქცია. ავიღოთ მასივი, ეს მასივი გავუგზავნოთ ფუნქციას არგუმენტის სახით და ამავდროულად გავუგზავნოთ მეორე არგუმენტად ის მეთოდი თუ რა მანერით მინდა რომ

დაამუშავოს პირველი არგუმენტი. მთავარ ფუნქციას ანუ „First class function-ს“ შედეგი დავაბრუნებინოთ და console.log-ით გამოვიტანოთ.

```
JS app.js > ...
1   ////////////////////////////
2   // Lecture: Passing functions as arguments
3
4   var years = [1990, 1965, 1937, 2005, 1998];
5
6   function arrayCalc(arr, fn) {
7       var arrRes = [];
8       for (var i = 0; i < arr.length; i++) {
9           arrRes.push(fn(arr[i]));
10      }
11      return arrRes;
12  }
13
14  function calculateAge(el) {
15      return 2019 - el;
16  }
17
18  function isFullAge(el) {
19      return el >= 18;
20  }
21
22  function maxHeartRate(el) {
23      if (el >= 18 && el <= 81) {
24          return Math.round(206.9 - (0.67 * el));
25      } else {
26          return -1;
27      }
28  }
29
30
31  var ages = arrayCalc(years, calculateAge);
32  var fullAges = arrayCalc(ages, isFullAge);
33  var rates = arrayCalc(ages, maxHeartRate);
34
35  console.log(ages);
36  console.log(rates);
37
```

ფუნქციის ეს შესაძლებლობა ნათლად გვაჩვენებს თუ როგორ ეფექტურად შეგვიძლია გამოვიყენოთ არგუმენტის სახით გატნეული „callback“ ფუნქცია. ეს მეთოდი ჩვენ უკვე გამოყენებული გვაქვს წინა ეტაპზე, როდესაც ღილაკის listening-ს მიმაგრებული გვქონდა ჩვენთვის სასურველი “callback” ფუნქცია.

5.7) First class function: functions returning functions

აქაც მსგავსად არის სიტუაცია, სადაც რაღაც პირობის ხარჯზე დავუბრუნებთ შედეგს სადაც შედეგი არის ფუნქცია. ჩვენს მაგალითში ვქმნით პირველი კლასის ფუნქციას ინტერვიუსთვის საჭირო კითხვების გენერატორს, თავისი არგუმენტით, რომელიც გვიბრუნებს შედეგად პროფესიის შესაბამის ფუნქციას ერთი არგუმენტით. პირველი კლასის ფუნქციის გამოძახებისას შესაბამისი პარამეტრით დაგვიბრუნებს შედეგს ამ შემთხვევაში ფუნქციას, რომელსაც ცვლადში ვათავსებთ (ეს მეთოდი მსგავსია Function Expresion-ის) და შემდგომ ამ ცვლადით უკეთებთ call-ს ფუნქციას და ვატნევთ პიროვნების სახელს პარამეტრის სახით. აქ მნიშვნელოვანი არის ფუნქციის გამოძახების უცნაური ფორმაც, რომელიც უპრობლემოდ იმუშავებს რადგან მარცხნიდან მარჯვნივ მიყვება . interviewQuestion(‘teacher’)(‘Tsotne’).

```
JS app.js > ...
1  ////////////////////////////
2  // Lecture: Functions returning functions
3
4  function interviewQuestion(job) {
5      if (job === 'designer') {
6          return function(name) {
7              console.log(name + ', can you please explain what UX design is?');
8          }
9      } else if (job === 'teacher') {
10         return function(name) {
11             console.log('What subject do you teach, ' + name + '?');
12         }
13     } else {
14         return function(name) {
15             console.log('Hello ' + name + ', what do you do?');
16         }
17     }
18 }
19
20 var teacherQuestion = interviewQuestion('teacher');
21 var designerQuestion = interviewQuestion('designer');
22
23
24 teacherQuestion('luka');
25 designerQuestion('luka');
26 designerQuestion('merabiko');
27 designerQuestion('eleniko');
28 designerQuestion('gio');
29
30 interviewQuestion('teacher')('davit');
31
32
```

5.8) Immediately Invoked Function Expression (IIFE)

IIFE javascript-ის ფუნქცია, რომელიც ეშვება მაშინვე როდესაც მისი აღწერა მოხდება. მას ასევე მოიხსენიებენ self-execution anonymous function სახელით და აქვს ორი მთავარი მახასიათებელი: პირველი ის რომ ანონიმური ფუნქცია არის “lexical scope”-ისთვის რომელიც ( )-ით განისაზღვრება (ანუ მასში აღწერილი ცვლადები იქნება ანონიმური) და მეორე ის რომ მყისიერად ხდება მისი execute-ი ფრჩხილების გამოყენებით ( )-ით.

(function () {
}) ();

ამ მეთოდით პრაქტიკაში global ცვლადები რომ არ გამოვიყენოთ და private გვქონდეს ამისათვის ვიყენებთ. მაგალითში კარგად ჩანს თუ როგორ იზოლირებულია „score“ ცვლადი „lexical scope“-ისგან რაც მის ამ ფუნქციის პრივატულ ხასიათზე მიუთითებს.

```
JS app.js > ...
1  ////////////////////////////
2  // Lecture: IIFE
3
4  function game() {
5      var score = Math.random() * 10;
6      console.log(score >= 5);
7  }
8  game();
9
10
11 (function () {
12     var score = Math.random() * 10;
13     console.log(score >= 5);
14 })();
15
16 //console.log(score);
17
18
19 (function (goodLuck) {
20     var score = Math.random() * 10;
21     console.log(score >= 5 - goodLuck);
22 })(5);
23
```

5.9) Closures

ჩვენ ძალიან ბევრი თეორიული ცოდნა მივიღეთ JS-ში, თუ როგორ მუშაობს და რა ლოგიკას იყენებს. ეხლა უკვე მზად ვართ და ყველაფერი გვაქვს იმისთვის რომ გავიგოთ „Closures“-ს მუშაობის პრინციპი რაც ძალიან მნიშვნელოვანი თემაა. იმისთვის რომ უკეთ გავიგოთ დავიხმარ პრაქტიკულ მაგალითს, სადაც შევქმნი პირველი კლასის ფუნქციას „retirment“ რომელსაც ექნება ერთი არგუმენტი „retirmentAge“. პირველი კლასის ფუნქციაში გვექნება ასევე დეკლარირებული ცვლადები და ბოლოს „return“-ით დავაბრუნებთ თავის მხრივ ანონიმურ ფუნქციას ერთი არგუმენტით „yerOfBirth“.

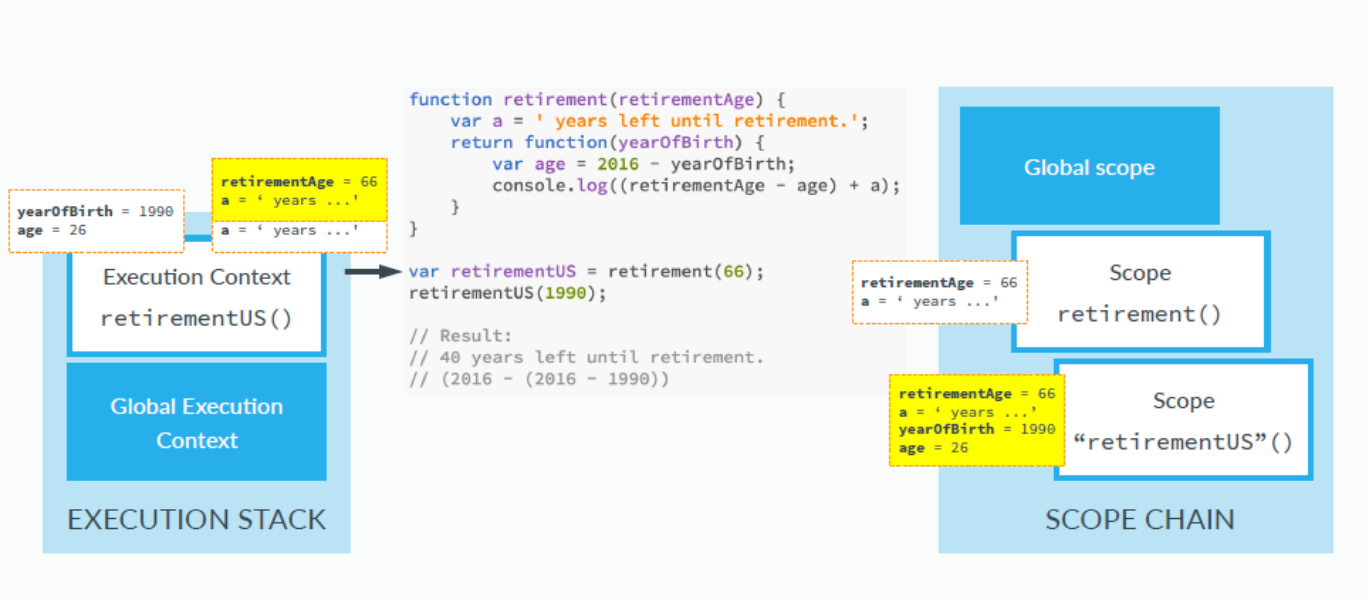
```
JS app.js > ...
1  ////////////////////////////
2  // Lecture: Closures
3
4  function retirement(retirementAge) {
5      var a = ' years left until retirement.';
6      return function(yearOfBirth) {
7          var age = 2016 - yearOfBirth;
8          console.log((retirementAge - age) + a);
9      }
10 }
11
12 var retirementUS = retirement(66);
13 var retirementGermany = retirement(65);
14 var retirementIceland = retirement(67);
15
16 retirementGermany(1990);
17 retirementUS(1990);
18 retirementIceland(1990);
19
20 //retirement(66)(1990);
21
22
23 function interviewQuestion(job) {
24     return function(name) {
25         if (job === 'designer') {
26             console.log(name + ', can you please explain what UX design is?');
27         } else if (job === 'teacher') {
28             console.log('What subject do you teach, ' + name + '?');
29         } else {
30             console.log('Hello ' + name + ', what do you do?');
31         }
32     }
33 }
34
35 interviewQuestion('teacher')('John');
36
```

ამ მაგალითში ყურადღება მიაქციეთ რომ უკვე დაბრუნებულ ფუნქციას შეუძლია მიწვდეს „retirmentAge“ ცვლადს, რომელიც პირველი კლასის ფუნქციას ეკუთვნის. გასაოცარია არა, მაგრამ ეხლა ავხსნათ თეორიულად რა ხდება და რა გვაძლევს ამ შედეგს.

CLOSURES SUMMARY

An inner function has always access to the variables and parameters of its outer function, even after the outer function has returned.

ალბათ გახსოვთ რომ ახალი ფუნქცია ქმნის ახალ „execution Context“-ს და ის მოხვდება „Execution Stack“-ის ზედა თაროზე. ჩვენ გვახსოვს, რომ „Execution Context“-ი არი Object-ი და მასში არი OV, Scope chain და this variable. ამიტომ ქვედა სურათზე მარცხენა მხარეს კარგად არის ნაჩვენები ის ცვლადები, რომელთანაც წვდომა შეეძლება „retirment“ ფუნქციას. „Scope Chain“-ი მოკლედ რომ ვთქვათ არის მიმთითებელი OV-ის მემორში. მას შემდეგ რაც პირველი კლასის ფუნქცია ანუ „retirment“-ი შესრულდება და დააბრუნებს შედეგს „Execution Stack“-იდან ამოიშლება და მასთან ერთად OV და Scope Chain-იც გაქრება ხო ? მაგრამ არა, „Closures“-ის საიდუმლოც ეს არის რომ მას შემდეგ რაც ფუნქცია დააბრუნებს შედეგს და ეს ფუნქციაც გაქრება „Execution Stack“-დან OV-ი ჯერ კიდევ აქ არის მემორიში და ხელმისაწვდომია „Scope Chain“-ის საშვალებით, შეგახსენებთ რომ ის მიმთითებელი არის OV-ის. ბოლოს კი „retirmentUS“ ფუნქციის გაშვებისას შეიქმნის თავის „Execution Context“-ს რომელიც თავის მხრივ შექმნის „OV“ და „Scope Chain“-ს, მაგრამ აქ ყურადღება მივაქციოთ ეს შიდა ფუნქციაა „retirment“-ის და შექმნის „Lexical Scope“-ს ანუ ის მიწვდება მშობლის „OV“-ს და ე ყველაფერი სურათში მარჯვენა მხარეს კარგად არის მოცემული.



„Closures“-ს გამოყენებით გავაკეთოთ ინტერვიუს ამოცანა დამოუკიდებლად !!!

5.10) Bind, Call and Apply

ჩვენ გვახსოვს რომ ფუნქცია არის ერთ-ერთი Object-ის ტიპის და ის გვაძლევს როგორც Object-ის კონსტრუირების საშულაბეას ასევე Array-ის შემთხვევაში გვაძლევს დამატებით მეთოდებს, რომელიც მემკვიდრეობით ერგო. დღევანდელ ლექციაში ჩვენ ვისაუბრებთ Call, Apply და Bind მეთოდებზე, რომლებიც this ცვლადის ხელით მითითების საშუალებას გვაძლევს. მოდი ამ ყველაფერს პატარა პრაქტიკული მაგალითით გაჩვენებთ. შემოვიტან „john“ ობიექტს არა ფუნქციის კონსტრუქტორის საშუალებით არამედ ჩვეულებრი ობიექტის ლიტერალით (ანუ ფიგურული ფრჩხილებით „{ }“). ამ ობიექტში განვსაზღვრავთ რამოდენიმე propert-ს და ასევე method-ს სახელად „presentation“ სადაც ფორმალური და მეგობრული მისალმების სცენა მექნება აღწერილი. ამ მეთოდს განუსაზღვრეთ ორი არგუმენტი „style“ და „timeOfDay“. დეკლარაციის შემდგომ თუ შევამოწმებთ ვნახავთ რომ მუშაობს მაგრამ ეხლა შემოვიტანოთ იგივე გზით ახალი ობიექტი „emily“ რომლის აღწერაში მეთოდი არ დავიტანოთ. რა უნდა ვქნათ თუ კი გვინდა „presentation“ მეთოდის გამოყენება ემილის ობიექტისთვის, რომელიც მის მახასიათებლებში არ არის. ამისათვის ჩვენ შეგვიძლია გამოვიყენოთ „Call“ მეთოდი, რომელსაც პირველ არგუმენტად ყოველთვის ეთითება „this“ ცვლად და ჩვენ შემთხვევაშიც მითითებული გვაქვს „emily“. რას ცვლის ეს ფაქტი ? ამ მანერით მოცემულ მეთოდში გამოყენებული „this“ უკვე იქნება ემილი ობიექტის მიმთითებელი და მუშაობისას განიხილავს მის მახასიათებლებს. გამოდის რომ ჩვენ დროებით ვისესხეთ ჯონის ობიექტიდან მეთოდი, ხელით განუსაზღვრეთ „this“ პარამეტრი და ამ ფუნქციისთვის ორი აუცილებელი არგუმენტი. ამ მეთოდის მსგავსი არის „Apply“-იც უბრალოდ მას „this“ არგუმენტის მითითების შემდეგ მასივის სახით ჭირდება უშუალოდ იმ ფუნქციის პარამეტრების მითითება რომელი მეთოდის გამოყენებასაც ვაპირებთ. ჩვენს მაგალითში საწყის ეტაპზე პარამატრების დეკლარაციას მასივის სახით არ ვაკეთებთ ამიტომ ეს მეთოდი არ იმუშავებს.

```
241  ////////////////////////////
242  // Lecture: Bind, call and apply
243
244  var john = {
245      name: 'John',
246      age: 26,
247      job: 'teacher',
248      presentation: function(style, timeOfDay) {
249          if (style === 'formal') {
250              console.log('Good ' + timeOfDay + ', Ladies and gentlemen! I\'m ' +
251              this.name + ', I\'m a ' + this.job + ' and I\'m ' + this.age + ' years old.');
252          } else if (style === 'friendly') {
253              console.log('Hey! What\'s up? I\'m ' +  this.name + ', I\'m a ' +
254              this.job + ' and I\'m ' + this.age + ' years old. Have a nice ' + timeOfDay + '.');
255          }
256      }
257  };
258
259  var emily = {
260      name: 'Emily',
261      age: 35,
262      job: 'designer'
263  };
264
265  john.presentation('formal', 'morning');
266
267  john.presentation.call(emily, 'friendly', 'afternoon');
268
269  //john.presentation.apply(emily, ['friendly', 'afternoon']);
270
271  var johnFriendly = john.presentation.bind(john, 'friendly');
272
273  johnFriendly('morning');
274  johnFriendly('night');
275
276  var emilyFormal = john.presentation.bind(emily, 'formal');
277  emilyFormal('afternoon');
```

ბოლოს კი ვისაუბროთ თუ როგორ მუშაობს „Bind“ მეთოდი, რომელიც Call მეთოდის მსგავსად this პარამეტრის გამოყენების საშუალებას გვაძლევს. თუმცა განსხვავება მათ შორის არის ის რომ Bind მეთოდი ფუნქციას არ უშვებს დაუყონებლივ არამედ აკეთებს მის ასლს და ამიტომაც ვინახავთ ცვლადში (Function Expresion-ის მსგავსად). ჩვენს მაგალითში კარგად ჩანს რომ ჯონის ობიექტიდან წამოღებული ფუნქცია შევინახეთ ცვლადში, სადაც წინასწარ (preset) განვსაზღვრეთ პარამეტრები და შემდგომ ჩვენს მიერ შანახულ ცვლადში ამ ფუნქციის გამოძახებისას აღარ დამჭირდება სრული პარამეტრების მითითება. ბოლოს კი გამოვიძახებთ ჩვენს შენახულ ფუნქციას დარჩენილი ერთი პარამეტრით, რომელიც იგივე შედეგს მოგვცემს. ის ტექნიკური შერულება რაც ჩვენ Bind მეთოდით გავაკეთეთ აქვს თავისი სახლე და ეძახიან „carrying“-ს. დასკვა „carrying“-ი არის ტექნიკური შერულება, რომლის დროსაც ვქმნით ფუნქციას სხვა ფუნქციის ბაზაზე წინასწარი პარამეტრების განსაზღვრით.

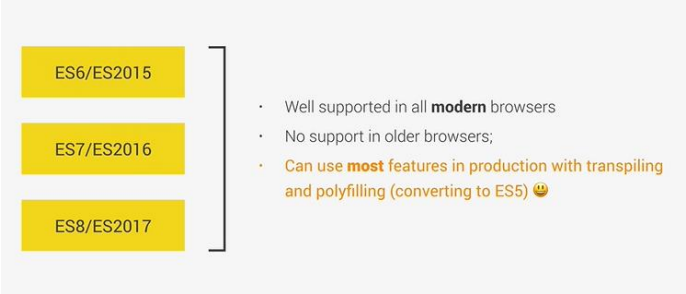
```
// Another cool example
var years = [1990, 1965, 1937, 2005, 1998];

function arrayCalc(arr, fn) {
    var arrRes = [];
    for (var i = 0; i < arr.length; i++) {
        arrRes.push(fn(arr[i]));
    }
    return arrRes;
}

function calculateAge(el) {
    return 2016 - el;
}

function isFullAge(limit, el) {
    return el >= limit;
}

var ages = arrayCalc(years, calculateAge);
var fullJapan = arrayCalc(ages, isFullAge.bind(this, 20));
console.log(ages);
console.log(fullJapan);
```

აქ მოცემულია კიდევ ჩვენთვის კარგად ნაცნობი მაგალითი სადაც გამოყენებული გვაქვს bind მეთოდი და კარგად დაგვანახებს მის პრაქტიკულ მხარეს. კონკრეტულად ამ მაგალითში ჩვენ „isFullAge“ ფუნქცია დავიყვანეთ ერთ არგუმენტზე რადგან თავის მხრივ ამ ფუნქციას პარამეტრის სახით იყენებს „arrayCalc“ ფუნქცია და მას დეკლარაციისას ერთი არგუმენტით აქვს აღწერილი „fn“ ფუნქციის მუშაობა.

Chapter 7 ES6/ES7 Next Generation Javascript

ამ თავში ჩვენ ES5-იდან გადავალთ ES6-ზე. Javascript Next Generation. ის რაც ვისწავლეთ რა თქმა უნდა ვალიდურია და ეხლაც მნიშვნელოვანია. ჩვენ განვიხილავთ იმ სიახლეებს რაც ახალ თაობას დაემატა, ძირითად სინტაქსურ ცვლილებებს და ა. შ. მომდევნო ლექციებში ჩვენ შევქმნით ახალ პროექტს javascript-ის ახალი თაობით.

7.1) what is new in ES6/ES2015



ES6-ის ვერსია არის ის პირველი თაობა, რომელშიც მოხდა უამრავი ცვლილება ES5-თან შედარებით. მას შემდეგ გამოვიდა ES7; ES8 და ა.შ. მაგრამ ჩვენ აქცენტს ვაკეთებთ ES6-ზე რადგან შემდეგ თაობებში რადიკალური განსხვავება არ გვაქვს. ჩვენ უკვე კარგად ვიცით რომ javascript-ის ახალი თაობის მხარდაჭერა აქვთ ახალ browser-ებს და იმისათვის რომ browser-ების ძველ თაობაზეც მორგებული იყოს უმეტესად იყენებენ transpiling და polyfilling-ის მეთოდს, რომლითაც ES6-ის კოდი გადაყავთ ES5-ში. ქვემოთ ჩამოთვლილი მაქვს ის სიახლეები რაც უნდა განვიხილოთ:

- Variable declaration with let and const
- Blocks and IIFFs
- Strings
- Arrow functions
- Destructuring
- Arrays
- The spread operator
- Rest and default parameters
- Maps
- Classes and subclasses

მოგვიანებით განვიხილავთ:

- Promises
- Native modules

ეს არის ძალიან მნიშვნელოვანი ორი საკითხი და მოგვიანებით გავივლით რადგან მათ ჭირდებათ წინასწარი მომზადება თუ რა როლი აქვს promises-ს ასინქრონულ javascript-ში და როგორ შეიძლება ჩვენი კოდი დავყოთ სხვადასხვა ფაილებში.

7.2) Variable declaration with let and const

ცვლადების დეკლარირებაში გვაქვს მნიშვნელოვანი ცვლილება, var-ის მაგივრად გვაქვს let ჩანაწერი და დამატებით გვაქვს const ჩანაწერი რომელიც ცვლადის მნიშვნელობას აღნიშნავს მუდმივ მნიშვნელობად, რომლის ცვლილება არ შეგვიძლია. ასევე ყველაზე მთავარი დეტალი let-ით და const-ით აღწერილი ცვლადები არიან block-scoped და არა function-scoped, რაც იმას ნიშნავს რომ { } ბლოკში შემოტანილი ცვლადი ბლოკის გარეთ უცნობია, რაც ჩვენ წინა ლექციებში ვახსენეთ. და ბოლოს ES6-ში execution context-ის დროს ცვლადების hoisting არ მუშაობს, რომლის ნათელი მაგალითია თუ კი ცვლადს გამოვიყენებთ declaration-ამდე გვექნება error როდესაც ES5-ში შედეგად გვექნებოდა undefined. მე ვფიქრობ რომ ES6-ის ეს ლოგიკა მართებულია რომ არ გამოგვეპაროს რაიმე შეცდომა.

```
////////////////////////////////
// Lecture: let and const

// ES5
var name5 = 'Jane Smith';
var age5 = 23;
name5 = 'Jane Miller';
console.log(name5);

// ES6
const name6 = 'Jane Smith';
let age6 = 23;
name6 = 'Jane Miller';
console.log(name6);


// ES5
function driversLicence5(passedTest) {

    if (passedTest) {
        console.log(firstName);
        var firstName = 'John';
        var yearOfBirth = 1990;
    }


    console.log(firstName + ', born in ' + yearOfBirth + ', is now officially allowed to drive a car.');
}

driversLicence5(true);


// ES6
function driversLicence6(passedTest) {

    //console.log(firstName);
    let firstName;
    const yearOfBirth = 1990;

    if (passedTest) {
        firstName = 'John';
    }

    console.log(firstName + ', born in ' + yearOfBirth + ', is now officially allowed to drive a car.');
}

driversLicence6(true);
```

7.3) blocks and IIFEs

წინა ლექციაში ჩვენ ვისაუბრეთ ცვლადების დეკლარირებაზე სადაც შემოვიტანეთ ტერმინი block-scoped; ეს ტერმინი აქაც გამოგვადგება რადგან data privace-ისთვის ჩვენ ვიყენებთ block-ს ES6-შიც. ანუ { }-ში გამოყენებული ცვლადები გარეთ არ ჩანს, რასაც ES5-ში ჩვენ IIFE-ის საშუალებით ვაკეთებდით (function () { })( ); Imediately Invoked Function Expression. აშკარად ES6-ში data privacy-ის პოლიტიკის გატარება მარტივად შეგვიძლია და აქ var-ზე მინდა გავამახვილო ყურადღება, რადგან Block-ში var-ით აღწერილი ცვლადის დანახვა block-ის გარეთაც შესაძლებელია რადგან არის function-scoped-ი.

```
64  ////////////////////////////////
65  // Lecture: Blocks and IIFEs
66
67  // ES6
68  {
69      const a = 1;
70      let b = 2;
71      var c = 3;
72  }
73
74  //console.log(a + b);
75  console.log(c);
76
77
78  // ES5
79  (function() {
80      var c = 3;
81  })();
82
83  //console.log(c);
84
```

7.4) Strings in ES6/ES2015

ES5-ში string-თან მუშაობაში გარკვეული გამოცდილება უკვე მივიღეთ, მაგრამ ES6-ში გვაქვს დამატებული string literals რომელსაც აქვს შემდეგი ფორმატი const n=`${firstname} ${lastname}`; სადაც firstname და lastname არის ცვლადები. ამ template-ის შედეგი n ცვლადში იქნება ერთი მთლიანი string სახელის და გვარის გაერთიანებით. ასევე string-ის method-ებშიც გვაქვს სიახლეები. მაგალითად: .startwith( ); .endwith( ); .includes( ); .repeat( ); მინდა კარგი მაგალითი გაჩვენოთ template-ის და method-ის ერთობლიობით:

```
90  ////////////////////////////////
91  // Lecture: Strings
92
93  let firstName = 'John';
94  let lastName = 'Smith';
95  const yearOfBirth = 1990;
96
97  function calcAge(year) {
98      return 2016 - year;
99  }
100
101 // ES5
102 console.log('This is ' + firstName + ' ' + lastName + '. He was born in ' + yearOfBirth + '. Today, he is ' + calcAge(yearOfBirth) + ' years old.');
103
104 // ES6
105 console.log(`This is ${firstName} ${lastName}. He was born in ${yearOfBirth}. Today, he is ${calcAge(yearOfBirth)} years old.`);
106
107
108 const n = `${firstName} ${lastName}`;
109 console.log(n.startsWith('j'));
110 console.log(n.endsWith('Sm'));
111 console.log(n.includes('oh'));
112 console.log(`${firstName} `.repeat(5));
113
```

console.log(`${name}_`.repeat(5));

თუ name არის Tsotne ამ ჩანაწერის შედეგი იქნება Tsotne_Tsotne_Tsotne_Tsotne_Tsotne დაიმახსოვრეთ: string-ის template literal-ის გამოყენება აუცილებლად რომელიც არის ` და არა ‘ ან „.

7.5) Arrow Functions: Basics

Arrow ფუნქციის სამი ძირითადი ტიპი გვაქვს, რომელი ES5-ში call back function ჩაანაცვლა ES6-ში, რომლის მაგალითი array.map-ის გამოყენებით ძალიან კარგად ჩანს. პირველ მაგალითში ვაძლევთ ერთ არგუმენტს, შედეგს აბრუნებს years.map(el => 2019 - el); მეორე მაგალითში უკვე ორ არგუმენტს ვაძლევთ string template-ის გამოყენებით ვაბრუნებ შედეგს: years.map((el index) => `age element ${index+1}:${2019-el}.`); და ბოლოს შეგვიძლია arrow ფუნქციას block-ი მივცეთ დავამუშავოთ შესაბამისად მოცემული არგუმენტები. ეს არის arrow ფუნქციის საბაზისო დეტალები. უფრო დეტალურად შემდეგ ლექციებში ავხსნით.

```
119  ////////////////////////////////
120  // Lecture: Arrow functions
121
122
123  const years = [1990, 1965, 1982, 1937];
124
125  // ES5
126  var ages5 = years.map(function(el) {
127      return 2016 - el;
128  });
129  console.log(ages5);
130
131
132  // ES6
133  let ages6 = years.map(el => 2016 - el);
134  console.log(ages6);
135
136  ages6 = years.map((el, index) => `Age element ${index + 1}: ${2016 - el}.`);
137  console.log(ages6);
138
139  ages6 = years.map((el, index) => {
140      const now = new Date().getFullYear();
141      const age = now - el;
142      return `Age element ${index + 1}: ${age}.`
143  });
144  console.log(ages6);
145
```

7.6) Arrow Functions: lexical ‘this’ keyword

ამ თავში ჩვენ ვისაუბრებთ თუ რატომ არის arrow ფუნქცია გამოყენებადი და ერთ-ერთი მისი დადებითი თვისება არის ის რომ, ის იზიარებს ‘this’ keyword-ს თავის ფუნქციაში. სხვა სიტყვებით რომ ვთქვათ მას არ აქვს საკუთარი ‘this’ და ის იყენებს ‘this’-რომელიც ეკუთვნის იმ ფუნქციას რომელშიც არის გამოყენებული => ეს arrow ფუნქცია. ასე რომ arrow ფუნქციას აქვს lexical ‘this’ variable. მაგალითისათვის მე ES5-ის სინტაქსით შემოვიტან object-ს, მასში აღვწერ მეთოდს(method)

```
////////////////////////////////
// Lecture: Arrow functions 2

// ES5
var box5 = {
    color: 'green',
    position: 1,
    clickMe: function() {

        var self = this; document.querySelector('.green').addEventListener('click', function() {
            var str = 'This is box number ' + self.position + ' and it is ' + self.color;
            alert(str);
        });
    }
}
//box5.clickMe();


// ES6
const box6 = {
    color: 'green',
    position: 1,
    clickMe: function() {
        document.querySelector('.green').addEventListener('click', () => {
            var str = 'This is box number ' + this.position + ' and it is ' + this.color;
            alert(str);
        });
    }
}
box6.clickMe();


const box66 = {
    color: 'green',
    position: 1,
    clickMe: () => {
        document.querySelector('.green').addEventListener('click', () => {
            var str = 'This is box number ' + this.position + ' and it is ' + this.color;
            alert(str);
        });
    }
}
box66.clickMe();
```

და ამ ფუნქციაში გამოვიძახებ კიდევ ფუნქციას სადაც ვიცით რომ 'this' ცვლადი მიუთითებს global object-ზე. რადგან ეს callback ფუნქცია არის regular function-ი, აქ გამოსავალი არის შემოვიტანოთ var self=this; ცვლადი რომელშიც შევინახავთ this-ს და ამის შემდეგ უკვე callback ფუნქციიდან მივწვდებით self-ს. რაც შეეხება ES6-ში უკვე მაგალითის შემთხვევაში თუ კი callback-ად => arrow ფუნქციას გამოვიყენებთ ვნახავთ რომ this variable-ს ჩვეულებრივ მივწვდებით რადგან arrow ფუნქცია იზიარებს lexical 'this' keyword-ს. აქ შეგვიძლია მოვიყვანოთ მეორე მაგალითი: object-ში მეთოდი რომელიც შემოვიტანეთ იყოს arrow და ამ arrow ფუნქციაში ჩავსვათ კიდევ arrow ფუნქცია, ამ შემთხვევაში this ცვლადს რომელსაც მივწვდებით იქნება global object-ის რადგან lexical ლოგიკით arrow ფუნქციამ თავიდანვე global object გაიზიარა და შესაბამისად საბოლოოდ arrow ფუნქცია ყოველთვის parent-ის ლოგიკას იზიარებს. Arrow ფუნქციის უპირატესობა აშკარად ჩანს აქ.

```
function Person(name) {
    this.name = name;
}

// ES5
Person.prototype.myFriends5 = function(friends) {

    var arr = friends.map(function(el) {
        return this.name + ' is friends with ' + el;
    }.bind(this));

    console.log(arr);
}

var friends = ['Bob', 'Jane', 'Mark'];
new Person('John').myFriends5(friends);


// ES6
Person.prototype.myFriends6 = function(friends) {

    var arr = friends.map(el => `${this.name} is friends with ${el}`);

    console.log(arr);
}

new Person('Mike').myFriends6(friends);
```

7.7) Destructing

Destructing-ი არის საშუალება რომელიც ამონაწერს გვაკეთებინებს object-იდან ან array-დან. ანუ თუ გვინდა რომ მასივიდან ამოვიღოთ რაღაც ინფორმაცია და განვათავსოთ ცალ-ცალკე ცვლადებში ამას ჩვენ ვაკეთებთ შემდეგნაირად: ['john', 26] name = array[0] და age = array[1] მაგრამ ES6-ში Destructor-ის საშუალებით ჩვენ შეგვიძლია შემდეგი სახით ჩავწეროთ: const [name, age] = [john, 30]; სადაც მასივის ელემენტები შესაბამისი იერარქიით აისახება name და age-ის ცვლადებში რომლისთვისაც ჩვეულებრივ გამოვიყენებთ console.log(name); იგივეს გაკეთება შეგვიძლია object-ზეც, მაგრამ ამ შეთხვევაშიუნდა გამოვიყენოთ { } ფიგურული ფრჩხილები და მასში მითითებული ცვლადის სახელები უნდა ემთხვეოდეს object-ის keyword-ს. წინააღმდეგ შემთხვევაში არ იმუშავებს const {name, age} = obj; თუ კი გვინდა ცვლადის სახელი განსხვავდებოდეს obj key-საგან მაშინ const { name:a, age:b} = obj; ეს მეთოდი ასევე შეგვიძლია გამოვიყენოთ როდესაც ფუნქციას შედეგად მასივს ვაბრუნებინებთ. მაგ.: const [age:retirement] = calcAgeRetirement(1989);

```
228  ////////////////////////////////
229  // Lecture: Destructuring
230
231  // ES5
232  var john = ['John', 26];
233  //var name = john[0];
234  //var age = john[1];
235
236
237  // ES6
238  const [name, age] = ['John', 26];
239  console.log(name);
240  console.log(age);
241
242  const obj = {
243      firstName: 'John',
244      lastName: 'Smith'
245  };
246
247  const {firstName, lastName} = obj;
248  console.log(firstName);
249  console.log(lastName);
250
251  const {firstName: a, lastName: b} = obj;
252  console.log(a);
253  console.log(b);
254
255
256
257  function calcAgeRetirement(year) {
258      const age = new Date().getFullYear() - year;
259      return [age, 65 - age];
260  }
261
262
263  const [age2, retirement] = calcAgeRetirement(1990);
264  console.log(age2);
265  console.log(retirement);
266
```

7.8) Arrays in ES6/ES2015

ამ თავში ვნახავთ თუ რა სიახლეები გვაქვს ES6-ში array-სთან მიმართებაში. პარალელს გავავლებთ ES5-თან და ვნახავთ როგორ გამარტივდა ES6-ში array-სთან მუშაობა. პირველ რიგში ვნახოთ თუ როგორ ვაკეთებთ list-ის გადაყვანას array-ში, კერძოდ slice მეთოდი call-ის გამოძახებით რისი გაკეთებაც ES6-ში .From ფუნქციით შეგვიძლია ანუ

```
272  ////////////////////////////////
273  // Lecture: Arrays
274
275  const boxes = document.querySelectorAll('.box');
276
277  //ES5
278  var boxesArr5 = Array.prototype.slice.call(boxes);
279  boxesArr5.forEach(function(cur) {
280      cur.style.backgroundColor = 'dodgerblue';
281  });
282
283  //ES6
284  const boxesArr6 = Array.from(boxes);
285  Array.from(boxes).forEach(cur => cur.style.backgroundColor = 'dodgerblue');
286
287  //ES5
288  for(var i = 0; i < boxesArr5.length; i++) {
289
290      if(boxesArr5[i].className === 'box blue') {
291          continue;
292      }
293      boxesArr5[i].textContent = 'I changed to blue!';
294  }
295
296  //ES6
297  for (const cur of boxesArr6) {
298      if (cur.className.includes('blue')) {
299          continue;
300      }
301      cur.textContent = 'I changed to blue!';
302  }
303
304  //ES5
305  var ages = [12, 17, 8, 21, 14, 11];
306
307  var full = ages.map(function(cur) {
308      return cur >= 18;
309  });
310  console.log(full);
311
312  console.log(full.indexOf(true));
313  console.log(ages[full.indexOf(true)]);
314
315  //ES6
316  console.log(ages.findIndex(cur => cur >= 18));
317  console.log(ages.find(cur => cur >= 18));
318
```

const boxes = document.querySelectorAll(‘.box’); ეს გვიბრუნებს nodelist-ს
const boxesArray = Array.from(boxes); list გადაგვყავს array-ში
boxesArray.forEach(cur => cur.style.backgroundColor = ‘red’);
ასევე array-სთან მიმართებაში გვაქვს ახალი loop-ის ოპერატორი for-of
for(const cur of boxesArray){ } სადაც შემიძლია continue და break-იც გამოვიყენოთ, რაც map-ის და foreach-ის შემთხვევაში არ გვქონდა. და ბოლოს გვაქვს ორი ახალი მეთოდი findIndex და find

console.log(ages.findIndex(cur => cur >= 18));
console.log(ages.find(cur => cur >= 18));
აქ ვიპოვით პირველი ელემენტის ინდექსს რომელიც ჩვენს პირობას აკმაყოფილებს (უშუალოდ ამ ელემენტს)

7.9) The Spread Operator

ამ თავში განვიხილავთ მეთოდს რომელიც მასივის ელემენტების ეფექტურად გამოყენების საშუალებას გვაძლევს. მაგალითად, როდესაც ფუნქციას გვინდა არგუმენტის სახით მივაწოდოთ რამოდენიმე ელემენტი, ამის გამარტივებული ვარიანტი არის თუ კი მასივს მივცემთ და არგუმენტებად გამოიყენოს array-ს ელემენტები. მაგალითად var x1 = sum(18,5,7,1); ფუნქცია ჩვეულებრივ იპოვის არგუმენტების ჯამს, მაგრამ ES5-ით ეს შეგვიძლია var number = [18,5,7,1]; მასივს არგუმენტის სახით მივაწვდით ფუნქციას var x2 = sum.apply(null, number); მაგრამ ეს ყველაფერი ბევრად მარტივი არის ES6-ში: const x3 = sum(…number); სადაც „...“ spread operator-ია და აკეთებს იგივეს, ანუ მასივის ელემენტებს შლის და არგუმენტების სახით აწვდის ფუნქციას. Spread operator შეგვიძლია ასევე გამოვიყენოთ ორი მასივის გასაერთიანებლად: Const number1 = [1, 2] და const number2 = [4, 5]; Spread operator-ის საშუალებით const all = […number1, …number2]; ასევე შეგვიძლია შუაში დამატებით კიდევ მივუთითოთ ელემენტი const all[…number1, 4, …number2];

```
////////////////////////////////
// Lecture: Spread operator


function addFourAges (a, b, c, d) {
    return a + b + c + d;
}

var sum1 = addFourAges(18, 30, 12, 21);
console.log(sum1);

//ES5
var ages = [18, 30, 12, 21];
var sum2 = addFourAges.apply(null, ages);
console.log(sum2);

//ES6
const sum3 = addFourAges(...ages);
console.log(sum3);


const familySmith = ['John', 'Jane', 'Mark'];
const familyMiller = ['Mary', 'Bob', 'Ann'];
const bigFamily = [...familySmith, 'Lily', ...familyMiller];
console.log(bigFamily);


const h = document.querySelector('h1');
const boxes = document.querySelectorAll('.box');
const all = [h, ...boxes];

Array.from(all).forEach(cur => cur.style.color = 'purple');

```

პრაქტიკულად როდესაც გვაქვს nodelist სადაც class-ის მიხედვით მაქვს ყველა ობიექტი და თუ გაქვს სხვა nodelist შეგვიძლია გავერთიანოთ და ერთად მოვახდინოთ მათი array-ში კონვერტაცია (რომელიც წინა ლექციაში განვიხილეთ) და forEach-ით ამ ელემენტების color-ის შეცვლას ერთიანად შევძლებთ.

7.10) Rest_Function

ამ ლექციაში ჩვენ განვიხილავთ function-ის პარამეტრებს და ჯერ ავხსნით რა არის rest პარამეტრები, რომელიც საშუალებას გვაძლევს თვითნებურად ფუნქციას მივაწოდოთ იმდენი არგუმენტი რამდენიც მინდა. Rest პარამეტრი spread ოპერატორის შებრუნებული ვარიანტი არის. ამ თავში ჯერ ვნახოთ ES5-ის ბაზაზე როგორ გავაკეთებთ ამ ყველაფერს და შემდეგ ვნახავთ ES6-ში rest პარამეტრის საშუალებით რამდენად მარტივად არის საქმე. ES5-ში მაქვს arguments რომელიც არის object და slice მეთოდით ის უნდა გადავიყვანოთ array-ში. ES6-ში კი გვაქვს ისევ „...“ rest პარამეტრი, რომელსაც ფუნქციის დეკლარაციისას ვიყენებთ function fullage(… years){ } years.foreach და გამოძახებისას ვაძლევთ ჩვენთვის სასურველ პარამეტრს. Rest პარამეტრი გამოიყენება function declaration-ის დროს ხოლო spread operator გამოიყენება function call-ის დროს. ეს არის მთავარი განსხვავება. Rest პარამეტრის მითითებისას შეგვიძლია დავამატოთ არგუმენტი ფუნქციის დეკლარაციის დროს.

```
369  ////////////////////////////////
370  // Lecture: Rest parameters
371
372  //ES5
373  function isFullAge5() {
374      //console.log(arguments);
375      var argsArr = Array.prototype.slice.call(arguments);
376
377      argsArr.forEach(function(cur) {
378          console.log((2016 - cur) >= 18);
379      })
380  }
381
382
383  //isFullAge5(1990, 1999, 1965);
384  //isFullAge5(1990, 1999, 1965, 2016, 1987);
385
386
387  //ES6
388  function isFullAge6(...years) {
389      years.forEach(cur => console.log( (2016 - cur) >= 18));
390  }
391
392  isFullAge6(1990, 1999, 1965, 2016, 1987);
393
394
395  //ES5
396  function isFullAge5(limit) {
397      var argsArr = Array.prototype.slice.call(arguments, 1);
398
399      argsArr.forEach(function(cur) {
400          console.log((2016 - cur) >= limit);
401      })
402  }
403
404
405  //isFullAge5(16, 1990, 1999, 1965);
406  isFullAge5(1990, 1999, 1965, 2016, 1987);
407
408
409  //ES6
410  function isFullAge6(limit, ...years) {
411      years.forEach(cur => console.log( (2016 - cur) >= limit));
412  }
413
414  isFullAge6(16, 1990, 1999, 1965, 2016, 1987);
415
```

7.11) Default_Function parameters

ამ თავში ჩვენ გავიხსენებთ Function constructor-ის საშუალებით როგორ უნდა შევქმნათ object მაგრამ აქ ყურადღებას გავამახვილებთ new ბრძანებით ფუნქციის გამოძახებისას პარამეტრების მითითებაზე, სადაც javascript არ გვაიძულებს, რომ ყველა პარამეტრი მივუთითოთ. მაგრამ იმ პარამეტრებს რასაც არ მივუთითებთ object-ში ის იქნება "undefined" და ის მომენტი რომ ES5-შ თუ მომხმარებელმა არ მიუთითა ყველა პარამეტრი default მნიშვნელობების შემოსატანად ვიყენებთ terner ოპერატორს ხოლო ES6-ში ეს ძალიან მარტივად არის. =>

```
421  ////////////////////////////////
422  // Lecture: Default parameters
423
424  // ES5
425  function SmithPerson(firstName, yearOfBirth, lastName, nationality) {
426
427      lastName === undefined ? lastName = 'Smith' : lastName = lastName;
428      nationality === undefined ? nationality = 'american' : nationality = nationality;
429
430      this.firstName = firstName;
431      this.lastName = lastName;
432      this.yearOfBirth = yearOfBirth;
433      this.nationality = nationality;
434  }
435
436
437  //ES6
438  function SmithPerson(firstName, yearOfBirth, lastName = 'Smith', nationality = 'american') {
439      this.firstName = firstName;
440      this.lastName = lastName;
441      this.yearOfBirth = yearOfBirth;
442      this.nationality = nationality;
443  }
444
445
446  var john = new SmithPerson('John', 1990);
447  var emily = new SmithPerson('Emily', 1983, 'Diaz', 'spanish');
448
449
```

7.12) Maps

ამ თავში განვიხილავთ ახალ საკითხს "Map"-ს რომელიც ES6-ში ჩანაწერებთან კომპლექსური მუშაობის საშუალებას იძლევა. Map არის object-ის მსგავსი მაგრამ მთლად ასე მასშტაბური არ არის. Map-ის უპირატესობა ის არის, რომ მასში გვაქვს forEach და მისი გამოყენება შეგვიძლია for-of-ში რაც object-ისაგან განასხვავებს.

Const question = new Map();

.set(); .get(); .size(); .has(); .clear(); .delete();

Map-ში გვაქვს Key და Value. key-ში შეიძლება იყოს ჩანაწერი, რიცხვი ან Boolean და value რა თქმა უნდა ნებისმიერი ტიპი. აქ განვიხილავთ მაგალითებს forEach-ის for-of-ის უბრალოდ for-of-ში უნდა გამოვიყენოთ: for(let [key, value] of question.entries()){ } destructing რომელიც შემოვიტანთ bracket-ს და განვსაზღვრავთ მასში key-ს და value-ს. შენიშვნა!!! parseInt() ფუნქცია მნიშვნელობას number-ში აკონვერტირებს.

```
////////////////////////////////
// Lecture: Maps

const question = new Map();
question.set('question', 'What is the official name of the latest major JavaScript version?');
question.set(1, 'ES5');
question.set(2, 'ES6');
question.set(3, 'ES2015');
question.set(4, 'ES7');
question.set('correct', 3);
question.set(true, 'Correct answer :D');
question.set(false, 'Wrong, please try again!');

console.log(question.get('question'));
//console.log(question.size);


if(question.has(4)) {
    //question.delete(4);
    //console.log('Answer 4 is here')
}

//question.clear();


//question.forEach((value, key) => console.log(`This is ${key}, and it's set to ${value}`));


for (let [key, value] of question.entries()) {
    if (typeof(key) === 'number') {
        console.log(`Answer ${key}: ${value}`);
    }
}

const ans = parseInt(prompt('Write the correct answer'));
console.log(question.get(ans === question.get('correct')));

```

7.13) Classes

Classe არის ერთერთი მთავარი სიახლე რაც ES6-მა შემოიტანა javascript-ის ცხოვრებაში და ამ მეთოდში ბევრად მარტივად ვახდენთ object-ის კონსტრუირებას, სანამ classe-ის პრაქტიკულ აღწერაზე გადავალთ მანამ ES5-ში განახებთ მსგავსი მუშაობის პრინციპს, რაც უკვე ძალიან კარგად გვესმის. Function Construction მეთოდით შევქმნათ object, მივუთითოთ პარამეტრები. შემდეგ prototype-ში დავამატოთ method და ბოლოს new ბრძანებით მოვახდიონოთ დეკლარირებული ფუნქციის გამოძახება.

ეხლა კი ES6-ის სინტაქსით classe გამოვიყენოთ. შემოვიტანოთ class Person{ }; ანუ მოვახდინეთ class-ის დეკლარაცია (ფუნქციის დეკლარაციის მსგავსად). ეხლა დავამატოთ მეთოდი constructor( ) რომელშიც განვსაზღვრავთ პარამეტრებს და this პარამეტრის ინიციალიზაციას მოვახდენ. ეს კონსტრუქტორი regular function-ს გავს. ალბათ თქვენ თვითონ ავლებთ პარალელს class-ის constructor-სა და ES5-ში function constructor-ში შესაბამის ბლოკს შორის. თუ კი class-ში დაგვჭირდება method-ის დამატება ამასაც მარტივად ვაკეთებთ. Class-ში method-ებს შორის არანაირი პუნქტუაციის ნიშნები არ გვჭირდება. და ბოლოს ამ class-ის ობიექტის ინიციალიზაცია ხდება იდენტურად const john6 = new Person(Tsotne, 1989, 'IT'); და თუ console-ში შევამოწმებთ john5-სა და john6-ს რომლებიც შესაბამისად ES5-სა და ES6-ში არის შექმნილი, ვნახავთ რომ ამ ორ object-ს შორის არანაირი განსხვავება არ არის. აქ არის მხოლოდ სინტაქსური განსხვავება რაც უფრო მარტივს ხდის class-ს.

```
496  ////////////////////////////////
497  // Lecture: Classes
498
499  //ES5
500  var Person5 = function(name, yearOfBirth, job) {
501      this.name = name;
502      this.yearOfBirth = yearOfBirth;
503      this.job = job;
504  }
505
506  Person5.prototype.calculateAge = function() {
507      var age = new Date().getFullYear - this.yearOfBirth;
508      console.log(age);
509  }
510
511  var john5 = new Person5('John', 1990, 'teacher');
512
513  //ES6
514  class Person6 {
515      constructor (name, yearOfBirth, job) {
516          this.name = name;
517          this.yearOfBirth = yearOfBirth;
518          this.job = job;
519      }
520
521      calculateAge() {
522          var age = new Date().getFullYear - this.yearOfBirth;
523          console.log(age);
524      }
525
526      static greeting() {
527          console.log('Hey there!');
528      }
529  }
530
531  const john6 = new Person6('John', 1990, 'teacher');
532
533  Person6.greeting();
534
```

დეველოპერების ზოგი ნაწილი ამ დეტალს აკრიტიკებენ რადგან class-ში დამალულია object-oriented ბუნება და inheritance. Class-ში შეგვიძლია შევქმნათ static method რომელიც გამოიძახება მხოლოდ class-ის სახელით.

7.14) Classes with Subclasses

ამ თავში ჩვენ გავაგრძელებთ საუბარს class-ებზე და კერძოდ ავხსნით თუ რა არის subclass ანუ სხვა სიტყვებით რომ ვთქვათ ორ class-ს შორის inheritance-ის გამოყენებას ES6-ში subclass-ით ვაკეთებთ, მაგრამ ჯერ ES5-ში გავაკეთოთ ეს ლოგიკა და შემდგომ შევადაროთ ES6-ის გადაწყვეტილებას.

ჩვენ წინა თავებში უკვე ვისაუბრეთ, სადაც გვქონდა Person5, function constructor-ით შექმნილი object და Athlete5, ასევე function constructor-ით შექმნილი object. შინაარსობრივად Athlete5 ამავდროულად არის ადამიანი რომელიც Person5-ის propert-ებს იზიარებს, ამიტომ ჩვენ უნდა გავაკეთოთ მისი inherit-ი ანუ ამ შემთხვევაში Athlete5-ს „გადააქვს“ Person5-ის მახასიათებლები თავის prototype-ში. ES6-ის ლოგიკით Athlete5 არის Subclass Person5-ის რადგან Person5 მოიცავს თავის მხრივ Athlete5-ის class-საც. ES5-ში function constructor-ით შევქმნათ ეს ორი object უბრალოდ Athlete5 რადგან არის subclass მის დეკლარაციაში გამოვიძახოთ Person5 ანუ Person5.call(this, name, birth); და მთავარი დეტალი თუ რატომ შემოვიტანეთ “this” ცვლადი ალბათ გახსოვთ new ოპერატორი, რომელიც ახალ

object-ს აგენერირებს, ამიტომ ეს this პარამეტრი ჭირდება რომ აღწერისას მის პარამეტრებთან გვქონდეს წვდომა, ამის შემდეგ გავაკეთებთ prototype chain-ს და ამას object create-ით გავაკეთებთ, რადგან ის გვაძლევს საშუალებას ხელით ჩავსვათ prototype object-ში. Athlete5.prototype = object.create(Person5.prototype); ამის შემდეგ ორი object გაერთიანდა და ეხლა შევქმნით ახალ Athlete5 ობიექტს.

```
540  ////////////////////////////////
541  // Lecture: Classes and subclasses
542
543  //ES5
544  var Person5 = function(name, yearOfBirth, job) {
545      this.name = name;
546      this.yearOfBirth = yearOfBirth;
547      this.job = job;
548  }
549
550  Person5.prototype.calculateAge = function() {
551      var age = new Date().getFullYear() - this.yearOfBirth;
552      console.log(age);
553  }
554
555  var Athlete5 = function(name, yearOfBirth, job, olymicGames, medals) {
556      Person5.call(this, name, yearOfBirth, job);
557      this.olymicGames = olymicGames;
558      this.medals = medals;
559  }
560
561  Athlete5.prototype = Object.create(Person5.prototype);
562
563  Athlete5.prototype.wonMedal = function() {
564      this.medals++;
565      console.log(this.medals);
566  }
567
568  var johnAthlete5 = new Athlete5('John', 1990, 'swimmer', 3, 10);
569
570  johnAthlete5.calculateAge();
571  johnAthlete5.wonMedal();
```

Var johnAthlet5 = new Athlete5(‘John’, 1990, ‘swimmer, 3, 10);

ამის შემდეგ თუ შევამოწმებთ ვნახავთ რომ johnAthlet5-ს prototype-ში Person5 რომლის ყველა property-ს გაიზიარებს და მაგალითისათვის გამოვიყენოთ johnAthlet5.calcAge( ); რომელიც Person5-ის method-ს გამოიყენებს.

```
573  //ES6
574  class Person6 {
575      constructor (name, yearOfBirth, job) {
576          this.name = name;
577          this.yearOfBirth = yearOfBirth;
578          this.job = job;
579      }
580
581      calculateAge() {
582          var age = new Date().getFullYear() - this.yearOfBirth;
583          console.log(age);
584      }
585  }
586
587  class Athlete6 extends Person6 {
588      constructor(name, yearOfBirth, job, olympicGames, medals) {
589          super(name, yearOfBirth, job);
590          this.olympicGames = olympicGames;
591          this.medals = medals;
592      }
593
594      wonMedal() {
595          this.medals++;
596          console.log(this.medals);
597      }
598  }
599
600  const johnAthlete6 = new Athlete6('John', 1990, 'swimmer', 3, 10);
601
602  johnAthlete6.wonMedal();
603  johnAthlete6.calculateAge();
604
```

რაც შეეხება ES6-ში subclass-ის შემოტანა ხდება: class Athlete6 extends Person6{ } ჩვეულებრივ subclass-საც ჭირდება constructor(name, birth, job, type, language){ } სადაც ცვლადების ინიციალიზაციას გავაკეთებ, მაგრამ subclass-ის ცვლადებს super(name, birth) ბრძანებით შემოვიტანთ. შემდეგ თუ რაიმე მეთოდი გვინდა შემოვიტანთ და ბოლოს შევქმნით ამ class-ის ობიექტს. Const johnAthlet6 = new Athlete6('John', 1990, 'swimmer, 3, 10); ამავდროულად მივწვდებით superclass-ის method-საც calcAge-ს. ბოლოს შევადაროთ ES5-ში შექმნილი ობიექტი ES6-ში შექმნილ ობიექტთან.

Chapter 8

Asynchronous Javascript: promises Async/

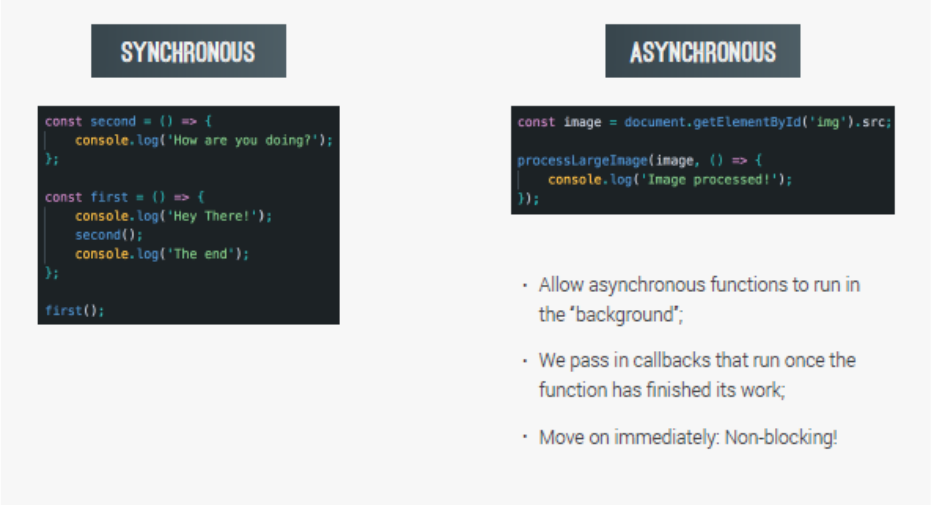
An example of Asynchronous Javascript

ამ თავში სტანდარტულად აღვწერ synchronous code-ს სადაც ES6-ის სინტაქსით შემოვიტან ფუნქციას, რომლის დეკლარაციას arrow function-ით გავაკეთებ, რომლის ძირითად ტანში, console-ის დახმარებით გამოვიტან რაიმე ტექსტს, ასევე მეორე ფუნქციასაც დავამატებ, რომლის დეკლარაციას ასევე Top level code-ში დავწერ და მის გამოძახებას მოვახდენ პირბელ ფუნქციაში. ამ სცენარით ჩვენ line-to-line შევქმნით Sync კოდს რომელიც სურათში კარგად ჩანს:

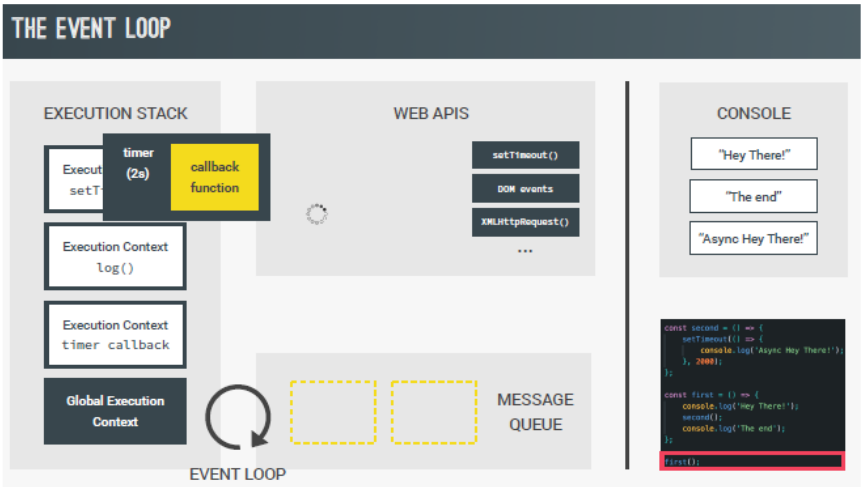
რაც შეეხება async კოდს აქ ხელოვნურად ტაიმერი მაქვს შემოტანილი მეორე ფუნქციაში რომელიც კონსოლში ტექსტის გამოტანას ჩვენს მიერ მიცემული დროით აგვიანებს და საბოლოო შედეგიც განსხვავებული გვაქვს. იმისთვის რომ ვმართოთ ასინქრონული სცენარი ჩვენს პროექტებში, პირველ რიგში ვნახოთ თავად Javascript-ი როგორ მუშაობს ამ დროს.

Asynchronous JS: Event loop

ზემოთ მოყვანილი Asynchronous კოდის მაგალითზე სიღრმისეულად ავხსნათ თუ რა ხდება ამ დროს კულისების უკან. მანამდე მინდა ვთქვა რომ აქამდე რაც გვიკეთებია ძირითადად სინქრონული სახე ქონდა ჩვენს კოდში, სადაც ყველა პირობა სრულდებოდა თანმიმდევრულად “line by line in a single thread in the javascript engine”.



ჩვენ წინა მაგალითში ხელოვნურად შემოვიტანეთ setTimeout()-ი რომელიც ჩვენ მიცემულ callback ფუნქციას უშვებდა გარკვეული დროის შემდეგ, მაგრამ მინდა გითხრათ რომ ამან არ შეაფერხა კოდის მსვლელობა და მითითებული დროის ამოწურვის შემდეგ callback ფუნქციაც შესრულდა. სანამ ახსნას დავიწყებ მინდა გითხრათ რომ user-ის მხარეს ხშირად დაგვჭირდება Asynchron-ული პროცესი, ეს იქნება geoLocation-ის დადგენა, DOM-ის მანიპულაციები, AJAX-ის სერვისები და ა.შ. მაგალითად შემიძლია მოვიყვანო DOM-ის Asynchron-ული სიტუაცია, როდესაც DOM-იდან შემოვიტანთ ფოტოს და ამ ფოტოს გავატნევთ ფუნქციას რომელიც რაღაც მანერით დაამუშავებს. ჩვენ ვიცით რომ ეს პროცესი გარკვეულ დროს საჭიროებს, მაგრამ ეს პერიოდი ჩვენ ვერ გავაჩერებთ კოდის execute-ს. თუ გავაჩერებთ წარმოიდგინეთ ღილაკზე ხელსაც ვერ დააჭერს user-ი. ამისათვის Asynchron-ული სიტუაცია ხშირად გვხვდება და ჩვენ ეხლა უნდა ავხსნათ კულისების უკან როგორ მართავს მსგავს სიტუაციას JS-ის კერძოდ V8 ძრავი. ამ შემთხვევაში chrome-ის მაგალითზე.



ჩვენი მაგალითის მიხედვით კოდის შესრულებისას მოხდება first ფუნქციის Execution stack-ში გადატანა სადაც შესრულდება console.log-ის ბრძანება. შემდეგ execution stack-ში შემოვა second( ) ფუნქცია რომელშიც setTimeout() method-ი დახვდება და როგორც გითხარით ის web API-ს ფუნქციებს ეკუთვნის, რომელიც javascript engine-ისგან განცალკევებული არის. მოცემული მეთოდი ჩართავს timer-ს და პარალელურად გაგრძელდება code-ის შესრულება და execution stack-ში first ფუნქციის ბოლო console.log ბრძანება შესრულდება. მაგრამ ზუსტად ეხლა შემოდის event loop-ის მცნება, როდესაც Timer ამოიწურება ეს callback ფუნქცია გადაინაცვლებს “Message Queu”-ში და event loop უზრუნველყოფს რიგრიგობით execution stack-ში გადმოტანას და შესრულებას. ამ პროცესის წარმოდგენაში სურათი დაგეხმარებათ. ეს არის ძალიან მნიშვნელოვანი საკითხი, რომელიც ყველა JS-ის developer-მა არ იცის. დასკვნა Event Loop მართავს Message Queu stack-ს და თანმიმდევრულად ასრულებს მასში არსებულ callback ფუნქციებს.

The Old way: Asynch Javascript with callbacks

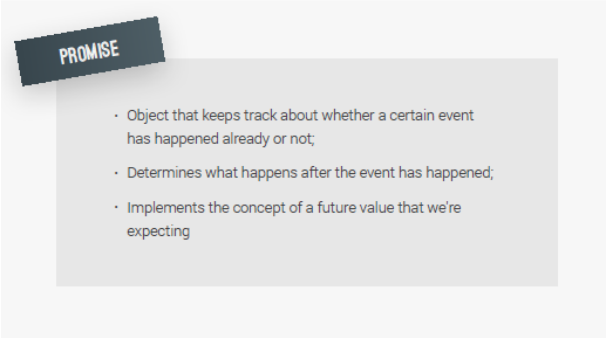
ჩვენ უკვე კარგად გვესმის ასინქრონული მუშაობის პრინციპი და დღევანდელ მაგალითში მე სიმულაციისათვის გამოვიყენებ setTimeout ფუნქციას რათა გავითამაშო AJAX-ის request response-ის სცენა. თითქოს და ჩვენ სერვერისაგან ვითხოვთ ID-ს შემდეგ ამ ID-ით ხელმეორე request-ით ვითხოვთ “recipes” და ბოლოს ვინც არის “რეცეპტის” publisher მისი კიდევ სხვა კერძი მოგვაქვს სერვერისაგან. სამივე მოთხოვნისათვის ჩვენ გამოვიყენებთ callback-ის ფუნქციას და ამ მაგალითში Asynchron-ული სიტუაცია ის არის რომ ჩვენ ზუსტად არ ვიცით server-იდან response რამდენ ხანში დაგვიბრუნდება. ამიტომ ქვემოთ სურათში კარგად არის მოცემული მთავარი ფუნქცია getRecipe( )-ის დეკლარაცია სადაც callBack-ით ID-ის მიხედვით მოგაქვს recipe და აქვე კიდევ შიგნით callback გვაქვს სადაც იმავე publisher-ის სხვა კერძი მოგვაქვს. მართალია ჩვენ ეს გავითამაშეთ setTimeout-ის დახმარებით მაგრამ რეალურადაც ესეთი სურათი გვაქვს. აქ მთავარი პრობლემა არის callback Hell რაც იმას გულისხმობს რომ callback-ების ჩადგმული ბლოკები გვაქვს, რომელიც სამკუთხედის ფიგურას ქმნის, რაც ძალიან დამამძიმებელი არის პროცესისათვის. რთულად სამართავიც

არის. ES6-ში ეს პრობლემა promise-ით აქვთ მოგვარებული, რასაც შემდეგ თავში განვიხილავთ.

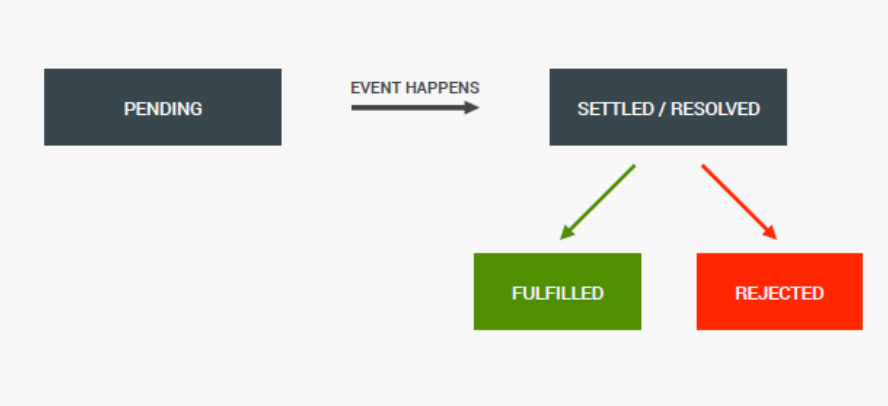
```
function getRecipe() {
    setTimeout(() => {
        const recipeID = [523, 883, 432, 974];
        console.log(recipeID);
            setTimeout(id => {
                const recipe = {title: 'Fresh tomato pasta', publisher: 'Jonas'};
                console.log(`${id}: ${recipe.title}`);
                    setTimeout(publisher => {
                        const recipe2 = {title: 'Italian Pizza', publisher: 'Jonas'};
                        console.log(recipe);
                    }, 1500, recipe.publisher);

            }, 1500, recipeID[2]);

    }, 1500);
}

getRecipe();
```

From Callback Hell to Promises

ამ თავში ჩვენ გავაგრძელებთ წინა თავის ამოცანას და კონკრეტულად callback hell-ის პრობლემას გადავწყვიტავთ ES6-ის ახალი "promise" მეთოდით. სანამ კოდის წერაზე გადავალთ მანამდე განვიხილოთ რა არის promise. ეს არის სპეციალურად Asynchron-ული JS-ისთვის შექმნილი object, რომელიც ინახავს და თვალყურს ადევნებს event-ებს, რომლებიც შეიძლება მოხდეს ან არა და თუ მოხდება შემდეგ არკვევს თუ რა უნდა მოხდეს შემდეგში.



Promise-ს შეუძლია მომავალში დაგიბრუნოს ის რასაც ელოდები. მაგალითად ჩვენ კულისების უკან სერვერისაგან მოვითხოვეთ რაიმე ინფორმაცია და "promise" გვპირდება რომ ამ data-ს მიიღებს და მომავალში შევძლებთ მის გამოყენებას. Promise როდესაც საქმეში ჩაერთვება მას აქვს სხვადასხვა "მდგომარეობა" (state), მაგალითად event სანამ მოხდება promise-ს აქვს "pending" სტატუსი, მას შემდეგ რაც event შესრულდება გადადის settled/Resolved სტატუსზე და ეს იმას ნიშნავს რომ მას ხელთ აქვს შედეგი (result) რომელიც შეიძლება იყოს წარმატებული და ვამბობთ promise არის "fulfilled" ან წარუმატებელი ანუ "rejected". სურათში კარგად ჩანს promise-ის ეს სტატუსები.



მოდი ეხლა ეს ყველაფერი პრაქტიკულად ვნახოთ კოდში. ჩვენ ამოცანაში სამი ფუნქციის შემოტანა მოგვიწევს რომელთაც new ბრძანებით promise-ის მეთოდში შევფუთავთ, სადაც callback ფუნქციაში გვექნება resolve და reject პარამეტრები, რომლებიც დაგვეხმარებიან საბოლოოდ fulfilled და error-ის განსაზღვრაში. მაგალითში კარგად ჩანს მათი გამოყენება. დეკლარაციის შემდეგ აუცილებელი არის ამ ფუნქციების გამოძახება, მაგრამ გამოძახებასთან ერთად მნიშვნელოვანი არის ხელი ჩავჭიდოთ სამომავლოდ შემოსულ result-ს, რომელსაც relosved() დამიბრუნებს. ამას then() მეთოდის საშუალებით შევძლებთ, რომელშიც event handler-ის დამატებით ხელს მოვკიდებ fulfilled-ს ანუ შედეგს (წარმატებულს), მაგრამ ამ დროს თუ რაიმე უზუსტობა მოხდა ანუ reject, ამ შემთხვევას catch( ) მეთოდით მოვკიდებ ხელს შესაბამისი event handler-ით.

```
const getIDs = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve([523, 883, 432, 974]);
    }, 1500);
});

const getRecipe = recID => {
    return new Promise((resolve, reject) => {
        setTimeout(ID => {
            const recipe = {title: 'Fresh tomato pasta', publisher: 'Jonas'};
            resolve(`${ID}: ${recipe.title}`);
        }, 1500, recID);
    });
};

const getRelated = publisher => {
    return new Promise((resolve, reject) => {
        setTimeout(pub => {
            const recipe = {title: 'Italian Pizza', publisher: 'Jonas'};
            resolve(`${pub}: ${recipe.title}`);
        }, 1500, publisher);
    });
};

getIDs
.then(IDs => {
    console.log(IDs);
    return getRecipe(IDs[2]);
})
.then(recipe => {
    console.log(recipe);
    return getRelated('Jonas Schmedtmann');
})
.then(recipe => {
    console.log(recipe);
})
.catch(error => {
    console.log('Error!!');
});
```

აქ ყურადღება მიაქციეთ ID-ის მიღების შემდეგ მეორე getRecipe( )-ის ფუნქციის გამოძახება then( ) მეთოდში არ გავაკეთოთ რადგან იგივე callback hell-ის ეფექტს მივიღებთ. ეს რომ ავიცილოთ თავიდან უნდა გამოვიყენოთ “chaining” რომლის საშუალებასაც “promise” გვაძლევს. ანუ return-ით დავაბრუნოთ და შემდეგ ეტაპზე “then”-ის შემოტანით ჩავჭიდებთ fulfilled-ს ხელს. სურათში კარგად ჩანს და ამ მეთოდით ჩვენ Asynchron-ულ პროცესთან მუშაობა ვისწავლეთ. უბრალოდ შემდეგ ლექციაში ვნახავთ ES8-ში თუ როგორ უფრო გაამარტივეს ეს ფუნქცია.

From Promeses to Async/Await

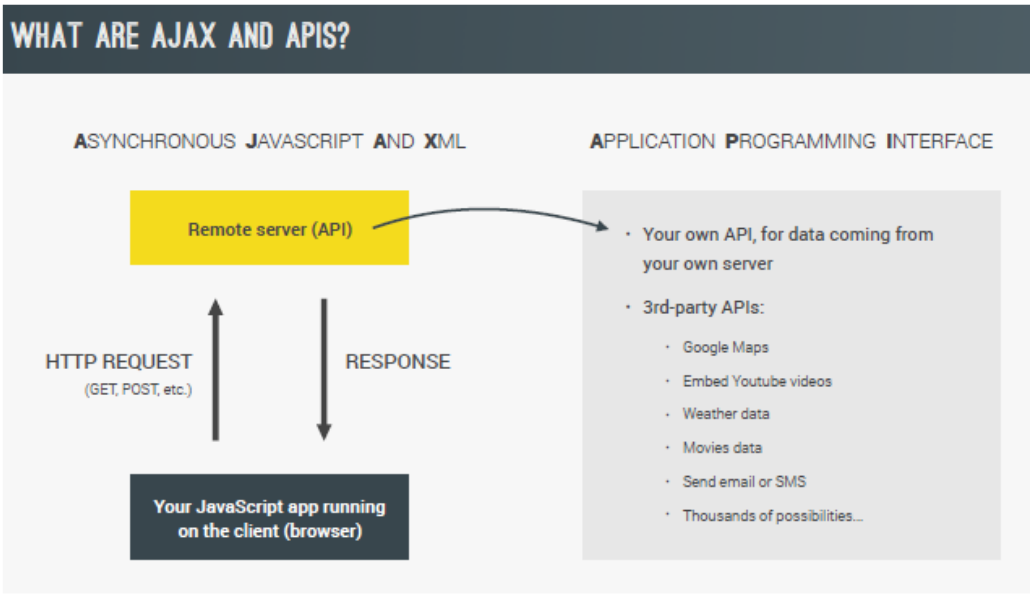
ჩვენ წინა თავში განვიხილეთ Promise-ის თემა, სადაც ხელოვნურად შექმნილი ასინქრონული პროცესის მიერ დაბრუნებული Fulfilled-ს და Reject-ს ვამუშავებდით. ეხლა კი მინდა გაგაცნოთ ES8/ES2017-ში Async/Await ფუნქცია რომელიც ბევრად გვიმარტივებს ასინქრონული პროცესების მართვას. ის რომ ჩვენ თავად შევქმნათ Promise ამას მოვახდენთ ჩვეულებრივ ძველი მანერით, მაგრამ then()-ის მაგივრად ჩვენ გამოვიყენებთ Async/Await. ამიტომ ძველი მაგალითით ვისარგებლებთ სადაც დეკლარირებულ Promise-ებს დავტოვებთ და ჩავანაცვლებთ then() ფუნქციას Async/Await-ით.

```
85  async function getRecipesAW() {
86      const IDs = await getIDs;
87      console.log(IDs);
88      const recipe = await getRecipe(IDs[2]);
89      console.log(recipe);
90      const related = await getRelated('Jonas Schmedtmann');
91      console.log(related);
92
93      return recipe;
94  }
95  getRecipesAW().then(result => console.log(`${result} is the best ever!`));
96
```

თავდაპირველად შემოვიტან async ჩანაწერს რომლითაც ფუნქციის დეკლრაციისას ხაზს უსვამ რაღაც კონკრეტულს. კერძოდ იმას რომ მოცემული ფუნქცია არ არის ჩვეულებრივი და ის შესრულებისას ავტომატურად გადავა background-ში და სხვა პროცესებს არ შეუშლის ხელს. ეხლა უკვე შეგვიძლია თანმიმდევრულად ხელი ჩავჭიდოთ ჩვენს მიერ დეკლარირებულ Promise-ებს, რომლისთვისაც შემოვიტანთ await ჩანაწერს. Await საშვალებას გვაძლევს „resolve“ შედეგს მოვკიდოთ ხელი და შედეგიც ჩვენთვის საურველ ცვლადში მოვათავსოთ. სურათზე კარგად ჩანს await-ის საშუალებით თუ როგორ ველოდებით promes-ს resolve შედეგს და შესაბამისად async ფუნქციაში მოცემული await ფუნქციის რიგითობა ერთი ერთში გადადის „Message queue“-ში და შესაბამისი რიგითობით სრულდება. ბოლოს კი შეგვიძლია თავად ასინქრონული ფუნქციის Promese-ის ჩავჭიდოთ ხელი then()-ის საშუალებით.

AJAX and API

AJAX-ი არის Asynchronous Javascript And Xml სერვისი რომელიც ასინქრონულად სერვერთან კომუნიკაციის საშუალებას გვაძლევს. ჩვენ შემთხვევაში კოდი რომელიც არის browser-ის მხარეს ვეძახით client-ს და იმისათვის, რომ სერვერისგან წამოვიღოთ ახალი მონაცემები საჭირო გვერდი განვაახლოთ. Ajax-ის საშუალებით ამ ყველაფერს ბევრად მარტივად გავაკეთებთ გვერდის reload-ის გარეშე.



სურათზე კარგად ჩანს რომ ajax-ის საშუალებით სერვერის მხარეს მოვახდენთ http request-ის გაგზავნას და მისგან response-ის მიღებას. აღსანიშანავია რომ ეს ყველაფერი ხდება ასინქრონულად background-ში რაც უკვე კარგად გვესმის. ამ მეთოდით ჩვენ შეგვიძლია სერვერიდან როგორც მივიღოთ ასევე გავაგზავნით კიდევაც. Ajax-ის სერვისის სიმულაცია ბევრნაირად შეგვიძლია მოვახდინოთ მათ შორის fetch მეთოდითაც. Client-Server-ულ კომუნიკაციაში აუცილებლად უნდა ავხსნათ თუ რა არის API (Application Programming Interface). სერვისი რომელიც გაშვებულია სერვერის მხარეს და გვაძლევს მონაცემთა გაცვლის საშუალებას აპლიკაციებს შორის. შეგვიძლია შევქმნათ ჩვენი საკუთარი API-ი ან ასევე შეგვიძლია გამოვიყენოთ შუამავლი API-ის სერვისები რომელიც სურათზე არის ჩამოთვლილი. API-ის გამოყენებით ბევრად გამარტივდა სრულფასოვანი სერვისების შექმნა.