JavaScript

Sedmi dio

Pregled

- Uvod
- Global Phase
- Izvršavanje funkcija
- Call stack
- Scope chain i outer environment
- Callbacks i Higher order functions
- Primjeri, PP
- Closures, prvi dio
- Primjeri, PP

Obnavljanje

- setTimeout, setInterval, clearInterval
- Cookies
- Window
- Hoisting
- Funkcije

Uvod

- U ovoj lekciji fokusiraćemo se više da teorijski dio JSa, pa ćemo uvesti neke nove pojmove
- **Syntax parser**
  - Program koji čita kod i provjerava šta taj kod radi
  - Takođe provjerava da li je gramatika ispravna (ovo ćete učiti iz kompajlera)
- **Lexical environments**
  - Gdje se kod fizički nalazi i šta se nalazi oko njega
  - Drugim riječima, vrlo je bitno gdje pišete kod
- **Execution contexts**
  - Wrapper oko određene cjeline koda i može da sadži neke stvari koje nisu vidljive kroz sam kod
  - Mi ćemo pominjati Local i Global execution context

Šta se dešava kad JS engine izvšava kod

```
const num = 3;
function multiplyBy2 (inputNumber){
  const result = inputNumber*2;
  return result;
}
const name = "Will"
```

- Odmah pri pokretanju programa kreira se **Global Execution Context**
  - Sadrži thread of execution (parsira i izvšava kod liniju po liniju)
  - Sadrži i dio za čuvanje podataka varijabli - **Global Variable Environment**
- Thread kod JSa
  - JS je single threaded (jedna stvar se izvšava u jednom trenutku)
  - Sinhrono izvršavanje (za sad samo to razmatramo)

Global Phase ili Global Execution Context

- Kada se kod izvrašava na baznom nivou (globalnom), JS engine radi sledeće stvari:
  - Kreira **globalni objekat**. Sav kod će se nalaziti u tom globalnom objektu
  - Kreira specijalnu varijablu **this.** Na globalnom nivou, this je globalni objekat **window**
  - Referencu, tj. vezu ka **outer environment-u** ako postoji
  - **Hoisting** pomoću kog se pripreme (sačuvaju) funkcije i kreira lista varijabli (sve setovane na **undefined**)
  - Onda kreće izvšavanje koda liniju po liniju (što uključuje dodavanje vrijednosti varijablama, čim se mijenja **undefined** vrijednost)
  - Obnavljanje, hoisting (code-4)

Izvršavanje funkcija

```
const num = 3;
function multiplyBy2 (inputNumber){
  const result = inputNumber*2;
  return result;
}

const output = multiplyBy2(4);
const newOutput = multiplyBy2(10);
```

- Kada se funkcija izvši kreira se novi **execution context** (local) koji se sastoji od:
  - **Thread of execution** (prolazimo kroz kod u funkciji liniju po liniju)
  - A local memory (**Local Variable Environment**) gdje se čuva sve što je definisano u funkciji

Call stack

- Služi nam za praćenje kojim redosledom funkcije trebaju da se izvše
  - Prati u kom Execution context-u se nalazimo
  - Prati koja se funkcija trenutno izvršava
  - Prati gdje trebamo da se vratimo nakon što se execution context izbaci iz steka
- Postoji jedan global execution context i više local execution context-a

Šta se dešava kada JS engine izvšava kod, pr-2

```
function a() {
        b();
        var c;
}
function b() {
        var d;
}
a();
var d;
```

```
function b() {
        var myVar;
        console.log(myVar);
}
function a() {
        var myVar = 2;
        console.log(myVar)
        b();
}
var myVar = 1;
console.log(myVar)
a();
console.log(myVar)
```

Scope chain i outer environment

- Šta se dešava sa sledećim programom?

```
function b() {
        console.log(myVar);
}
function a() {
        var myVar = 2;
        b();
}
var myVar = 1;
a();
```

```
function a() {
        function b() {
                console.log(myVar);
        }
        var myVar = 2;
        b();
}
var myVar = 1;
a();
```

- Šta ako u function a() izmijenimo redosled za myVar i b() ?

Callbacks, motivacioni primjer

- Kako bi ste napisali funkciju koja računa kvadrat broja 5
- -II- broja 6
- Itd.
- Šta je problem? Koji koncept je narušen? Kako rješavamo ovaj problem?

```
function copyArrayAndMultiplyBy2(array) {
  let output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] * 2);
  }
  return output;
}
const myArray = [1,2,3]
let result = copyArrayAndMultiplyBy2(myArray)
```

```
function copyArrayAndDivideBy2(array) {
  let output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] /2);
  }
  return output;
}
const myArray = [1,2,3]
let result = copyArrayAndDivideBy2(myArray);
```

- Kako biste napisali funkciju koja svakom elementu niza dodaje 3 i vraće taj novi niz?
- Imamo isti problem

Callbacks kao rješenje

- Možemo da generalizujemo našu funkciju tako što proslijedimo specifičnu instrukciju samo kad pokrenemo glavnu funkciju

```
function copyArrayAndManipulate(array, instructions) {
  let output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(instructions(array[i]));
  }
  return output;
}


function multiplyBy2(input) {
  return input * 2;
}


let result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

Zašto prethodni primjer radi

- Funkcije kod JSa su first-class objekti što znači da se funkcije mogu tretirati kao objekti
  - Mogu biti dodijeljene varijabli (function expression) ili property kao objektu
  - Mogu biti argumenti drugih funkcija (Callbacks)
  - Mogu biti vraćene iz druge funkcije (dio Closures)

Higher-order function vs callback

- **Higher-order function**
  - Tip funkcije koja ima drugu funkciju kao argument ili vraće neku novu funkciju
- **Callback**
  - Tip funkcije koji predstavlja argument druge funkcije

```
function copyArrayAndManipulate(array, instructions) {
  let output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(instructions(array[i]));
  }
  return output;
}


function multiplyBy2(input) {
  return input * 2;
}


let result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

Primjeri

- http://csbin.io/callbacks
  - Challenge 1 - Challenge 4
- Za domaci ćete trebati da odradite Extension 1 - Extension 6
  - Ne morate da koristite reduce()

Closures

- "*Closure is when a function remembers its lexical scope even when the function is executed outside that lexical scope*"
- Kada je funkcija pozvana, kreira se local variable environment (live store of data/local memory state) za function execution context funkcije koja je pozvana
- Kada funkcija završi izvršavanje, local variable environment se obriše zajedno sa execution context-om
- Ali šta ako funkcija ima način da sačuva podatke iz local variable env. između izvršavanja
- Ovo bi omogućilo definiciji funkcije da ima određeni cache/persistent podacima
- Ovo možemo realizovati tako što prvo omogućimo da funkcija vrati definiciju druge funkcije
- Primjenjuje se najviše kod module pattern-a

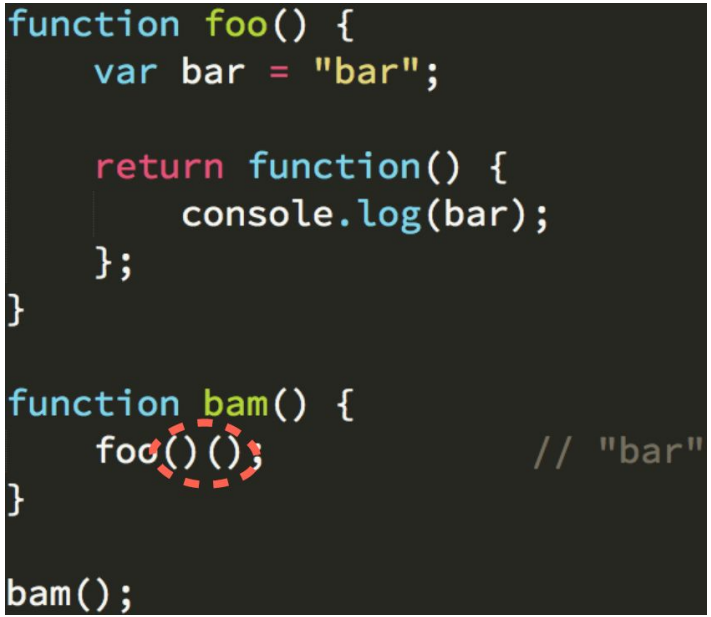
Closures, primjer

```
function instructionGenerator() {
  function multiplyBy2 (num){
    return num*2;
  }
  return multiplyBy2;
}

let generatedFunc = instructionGenerator()

let result = generatedFunc(3) //6
```

Closures, primjer-1



Closures, primjer-2

```
function greet(whattosay) {
        return function(name) {
                console.log(whattosay + ' ' + name);
        }
}

var sayHi = greet('Hi');
sayHi('Tony');
```

Closures, primjer-3

- U zavisnosti od toga gdje se definiše funkcija određuje se kojim varijablama funkcija ima pristup kad bude pozvana

```
function outer (){
  let counter = 0;
  function incrementCounter (){
    counter ++;
  }
  incrementCounter();
}

outer();
```

- Šta ako nakon outer pozovemo increaseCounter() ?

Closures, primjer-4

- Pri definisanju funkcije, za tu funkciju zakači se [[scope]] property koji referencira ka local variable environment u kome je funkcija definisana

```
function outer (){
  let counter = 0;
  function incrementCounter (){
    counter ++;
  }
  return incrementCounter;
}

let myNewFunction = outer(); // myNewFunction = incrementCounter
myNewFunction();
myNewFunction();
```

- Kad god pozovemo incrementCounter funkciju, prvo će biti provjeren trenutni local variable environment, a onda [[scope]] property, a potom sve ostalo

Static/Lexical scoping

- U prethodnom primjeru vidjeli smo čemu sliži i gdje se još primjenjuje Lexical Scope
- Naš lexical scope (raspoloživi podaci kad se definiše funkcije) određuje raspoložive varijable i prioritet pri izvršavanju funkcije, a ne mjesto gdje se onda izvrši. Šta ako ako opet pozovemo outer

```
function outer (){
  let counter = 0;
  function incrementCounter (){
    counter ++;
  }
  return incrementCounter;
}


let myNewFunction = outer();
myNewFunction();
myNewFunction();

var anotherFunction = outer(); // myNewFunction = incrementCounter
anotherFunction();
anotherFunction();
```

Primjeri

- http://csbin.io/closures
  - Challenge 1- Challenge 4
- Za domaći ćete trebati da odradite Challenge 5, Challenge 6, Challenge 7

Pitanja