JavaScript

Osmi dio

Pregled

- Closures, uvod
- PP
- Async (timers) under the hood
- PP

Obnavljanje

- Da li se parametri funkcije interno deklarišu sa var, let ili const?
- var, let i const hoisting, samo var podržava hosting?
- Neko da opiše global create phase
- Objasniti local execution context
  - Od čega se sastoji, kad se kreira
- Call stack i global execution context
- Šta su callback, a šta higher order funkcije
- Koje su prednosti callback funkcije, pogledajmo primjer sa prethodnog časa
  - Slajdovi 11 i 12
- Šta smo rekli za outer environment

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

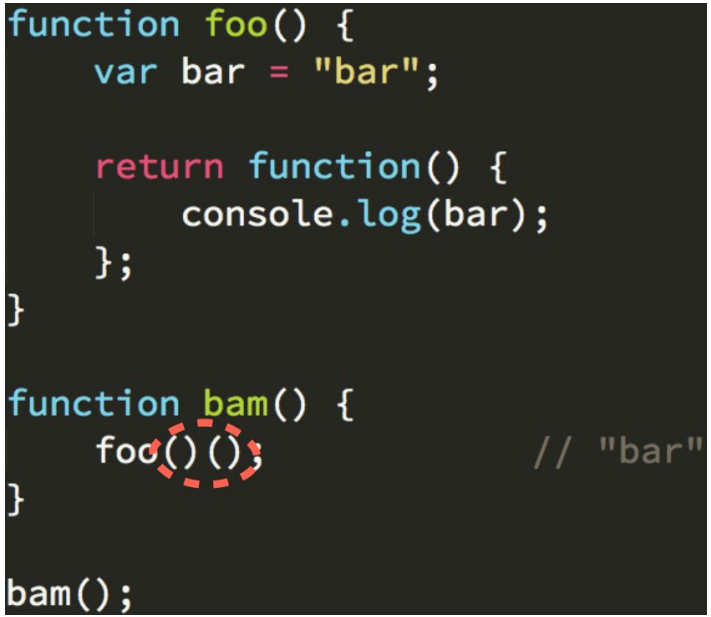
Closures, primjer

```
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

Uvod u asinhronost

- JS je single threaded (jedna komanda se izvršava u jednom momentu) i sadrži sinhroni izvršni model (svaka linija se izvršava u redosledu kako se kod pojavljuje)
- Ali šta ako je potrebno da prođe određeno vrijeme da bi se izvršio određeni dio koda?
  - Npr. potrebno je da sačekamo dok podaci stignu sa servera (API/server request)
  - Npr. timer da istekne i nakon toga da se izvši dio koda
    - Već vidjeli, setTimeout() i setInterval()
- Ključno pitanje, da li želimo da odložimo izvršavanje koda na određeni period ali da ne blokiramo thread da izvršava neki drugi dio koda. Tada izvršavanje koda postaje asinhrono i otežava nam praćenje izvršavanja koda

Primjeri

```
function printHello(){
    console.log(“Hello”);
}

setTimeout(printHello,1000);

console.log(“Me first!”);
```

No blocking!?

- Naš prethodni model ovdje ne radi !
- Moramo da proširimo naš model

```
function printHello(){
    console.log(“Hello”);
}

setTimeout(printHello,0);

console.log(“Me first!”);
```

Proširenje modela, nova terminologija

- Do sad smo uveli
  - Thread of execution
  - Memory/variable environment
  - Call stack
- Trebamo da uvedemo nove termine
  - Web Browser APIs background threads
  - Callback/Message queue
  - Event loop

Primjer

```
function printHello(){
    console.log(“Hello”);
}

function blockFor1Sec(){
    //blocks in the JavaScript thread for 1 second
}

setTimeout(printHello,0);

blockFor1Sec()

console.log(“Me first!”);
```

- Potrebno je da razumijemo kako spoljni svijet komunicira sa našim JS execution modelom. Šta će se desiti ovdje?

Izvršavanje asinhronog koda

- Dva glavna pravila za izvršavanje asinhronog koda
  - Čuvamo svaku “odloženu” funkciju u **Callback Queue** kada se završi **API thread**
  - Dodamo tu funkciju iz callback queue u call stack samo kada je call stack potpuno prazan
    - **Event loop** provjerava ovaj uslov
- Postoji puno stvari gdje čekanje može blokirati naš thread, pa koristimo Browser API
  - Timer
  - Nove informacije sa servera
  - Indikacija da je dio stranice učitan
  - Korisnička interakcija (clicks, mouseovers, drags, ...)
  - Writing/Reading u file sistem (Node)
  - Writing/Reading baze (Node)

Primjeri

- Primjeri, PP
  - http://csbin.io/async

Pitanja