JavaScript

Petnaesti dio

Pregled

- Objektno orijentisano programiranje
  - Uvod
  - Primjer
  - Pristupi kreiranja objekta
  - _ _proto_ _
  - Prototype
  - new keyword
  - This i arrow funkcije
  - ES6 pristup
  - call(), apply(), bind()
  - this

Obnavljanje

- Šta smo rekli za funkcionalno programiranje
  - Pure functions
  - Performanse
  - Prednosti
  - Koje funkcije se najviše koriste u praksi, a dio su funkcionalnog programiranja
  - Lodash

OOP, uvod

- Vrlo popularna paradigma za struktuiranje kompleksnog koda
  - Jednostavno dodavanje osobina i funkcionalnosti
  - Jednostavno za nas i druge developere da razmišljaju (jasna struktura)
  - Dobre performanse (efikasno što se tiče memorije)
- Potrebno je organizovati kod tako da ne bude sastavljen samo od niza komandi već da predstavlja neku organizovanu cjelinu

OOP, primjer

- Pretpostavimo da pravimo kviz igru
- Nek imamo sledeće korisnike:
  - Name: Phil
  - Score: 5
  - Name: Julia
  - Score: 4
  - **Funkcionalnost**: Mogućnost uvećanja Score-a
- Koji je najbolji način za čuvanje ovih podataka i funkcionalnosti?

OOP, primjer, nastavak

```
const user1 = {
  name: "Phil",
  score: 4,
  increment: function() {
    user1.score++;
  }
};

user1.increment(); //user1.score => 4
```

- Čuvanje funkcija sa njihovim povezanim (associated) podacima
  - Princip enkapsulacije

OOP, još funkcionalnosti za primjer

- U realnosti, imali bismo puno različitih funkcionalnosti povezanih sa user objektima
  - Mogućnost da smanjimo score
  - Mogućnost brisanja korisnika
  - Logovanje korisnika
  - Logout korisnika
  - Dodavanje avatara
  - Učitavanje podataka o korisniku
  - Još mnogo drugih funkcionalnosti

OOP, primjer, drugi način kreiranja

```
const user2 = {}; //create an empty object

user2.name = "Julia"; //assign properties to that object
user2.score = 5;
user2.increment = function() {
  user2.score++;
};
```

OOP, primjer, treći način kreiranja

```
const user3 = Object.create(null);

user3.name = "Eva";
user3.score = 9;
user3.increment = function() {
  user3.score++;
};
```

- Kod počinje da se ponavlja, kršimo DRY koncept
- Zamislite da imamo milion korisnika!
- Kako ovo da ispravimo?
- Šta god da stavimo kao argument za Object.create(), user3 će biti empty object, ali vidjećemo kasnije čemu će user3 dobiti pristup ako proslijedimo neke korisne podatke

OOP, primjer, kreiranje objekta, 4. pristup

```
function userCreator(name, score) {
  const newUser = {};
  newUser.name = name;
  newUser.score = score;
  newUser.increment = function() {
    newUser.score++;
  };
  return newUser;
};

const user1 = userCreator("Phil", 4);
const user2 = userCreator("Julia", 5);
user1.increment()
```

OOP, mane prethodnog pristupa

- Svaki put kada kreiramo novog usera, kreira se prostor u memoriji na računaru za sve podatke i funkcije. Ali ove funkcije su samo kopije !
- Postoji li bolji način?
- Prednost ovog pristupa je jednostavnost

OOP, peti pristup, prototype

- Čuvanje increment() funkcije u jednom objektu
- Kako da linkujemo sve objekte na tu funkciju?
  - **Prototype chain**

```
const functionStore = {
  increment: function(){this.score++;},
  login: function(){console.log("You're loggedin")}
};

const user1 = {
  name: "Phil",
  score: 4
}


user1.name // name is a property of user1 object
user1.increment // Error! increment is not!
```

OOP, primjer, prototype pristup(nastavak)

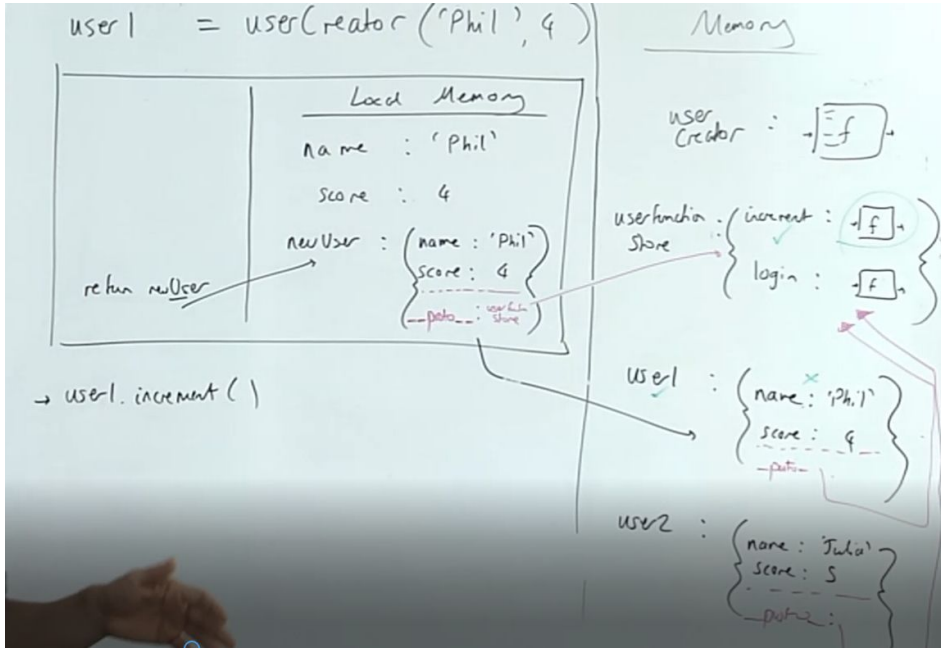
```
function userCreator (name, score) {
  const newUser = Object.create(userFunctionStore);
  newUser.name = name;
  newUser.score = score;
  return newUser;
};


const userFunctionStore = {
  increment: function(){this.score++;},
  login: function(){console.log("You're loggedin");}
};

const user1 = userCreator("Phil", 4);
const user2 = userCreator("Julia", 5);
user1.increment();
```

- __proto__ object hidden property, rereferenca

OOP, prethodni primjer, objašnjenje



OOP,prethodni primjer,prototype pristup,mane

- Nema većih problema
- Možda previše koda

```
const newUser = Object.create(functionStore)

    ....

return newUser
```

- Ovo moramo svaki put da pišemo - mada i nije neki problem, samo 6 riječi !
- Sofisticirano, ali nije standard

OOP, šesti pristup, keyword new

- const user1 = new userCreator(“Phil”, 4)
- Kada pozovemo konstruktor funkciju sa new ispred poziva te funkcije, automatizujemo dvije stvari
  - Kreiranje novog user objekta
  - Vraćanje novog user objekta
- Ali sada moramo da podesimo kako pišemo tijelo funcije userCreator:
  - Kako možemo sada da napravimo referencu ka automaski kreiranom objektu
  - Kako možete sada napraviti referencu ka funkcijama objekta kog smo kreirali

Funkcije su takođe i objekti

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

multiplyBy2.stored = 5
multiplyBy2(3) // 6

multiplyBy2.stored // 5
multiplyBy2.prototype // {}
```

- Pričali smo još na početku da je je kod JS sve objekat, pa i funkcije. Iz te činjenice funkcije mogu da sadrže i svojstva (properties). Iskoristićemo činjenicu da sve funkcije imaju default property, “prototype” - objekat. Pogledajmo i console gdje se nalaze ta skrivena svojstva

OOP, standardni pristup

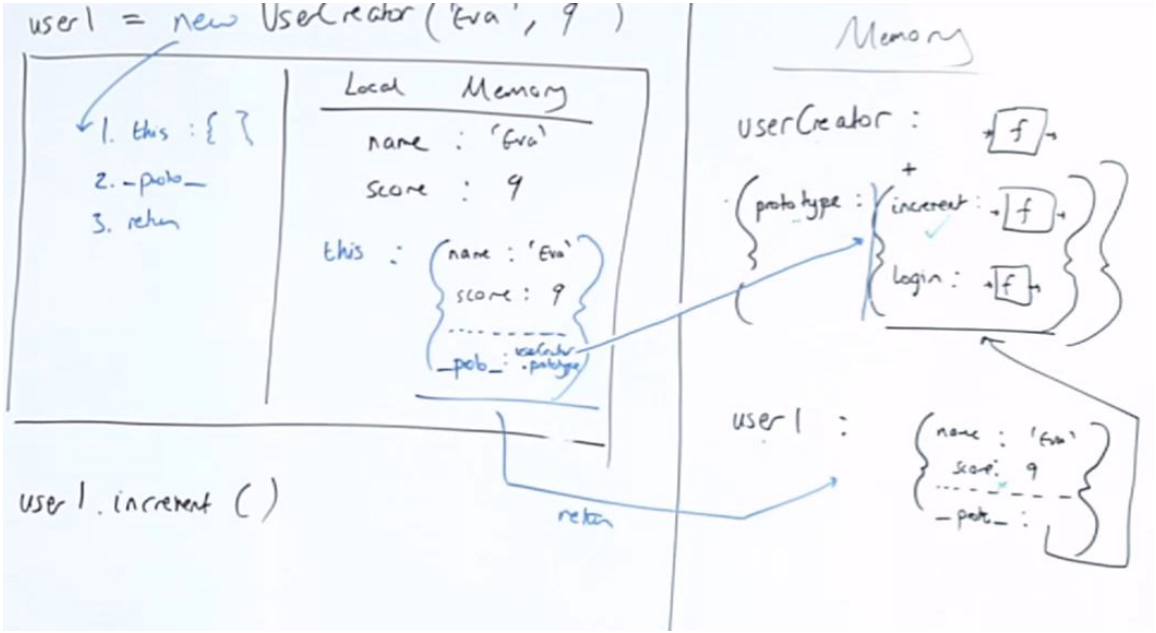
```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  this.score++;
};
UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator(“Eva”, 9)

user1.increment()
```

OOP, objašnjenje prethodnog primjera



OOP, prednosti i mane prethodnog pristupa

- Prednosti
  - Brže za implementaciju
  - Tipična praksa za pisanje OOP koda
- Mane
  - Veliki broj developera nema ideju kako radi ovaj pristup
  - Trebamo da stavimo prvo slovo funkcije veliko da na neki način sugerišemo drugim developerima da ispred ovog tipa funkcije treba da stave new keyword

Organizovanje koda u jednu shared funkciju

```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  function add1(){
    this.score++;
  }
  // const add1 = function(){this.score++;}
  add1()
};

UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator("Eva", 9)

user1.increment()
```

Šta je problem sa ovim?
Šta se dešava sa this unutar funkcije add1()?
Funkcije unutar metoda i this, problem
Često se koristi var that = this prije poziva metoda add1, da bismo sačuvali referencu ka this objektu !
Neki od način da se riješi ovaj problem osim ovog što smo već naveli:

- Koristimo bind, call, apply (vidjećemo kasnije kako)
- Koristimo arrow funkcije

OOP, this i arrow funkcije

```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  const add1 = ()=>{this.score++}
  add1()
};

UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator(“Eva”, 9)

user1.increment()
```

Kod arrow funkcija, this ima istu vrijednost kao i u funkciji u kojoj je **arrow** funkcija deklarisana !

Arrow funkcije su lexically scoped - bitno gdje smo definisali funkciju, a ne gdje smo je pozvali

OOP, ES6 pristup, klase

```
function UserCreator(name, score){
  this.name = name;
  this.score = score;
}

UserCreator.prototype.increment = function(){
  this.score++;
};
UserCreator.prototype.login = function(){
  console.log("login");
};

const user1 = new UserCreator(“Eva”, 9)

user1.increment()
```

```
class UserCreator {
  constructor (name, score){
      this.name = name;
      this.score = score;
  }
  increment (){
    this.score++;
  }
  login (){
    console.log("login");
  }
}

const user1 = new UserCreator("Eva", 9);

user1.increment();
```

Identično, jednostavnija sintaksa, syntactic sugar

call(), apply()

```
const obj = {
  num: 3,
  increment: function(){this.num++;}
};

const otherObj = {
  num: 10
};

obj.increment(); // obj.num now 4

obj.increment.call(otherObj); // otherObj.num now 11
// obj.increment.apply(otherObj);
```

- U ovom primjeru, **call**(*this_reference*) i **apply**(*this_reference*) se isto izvršavaju, ali u slučaju da increment(args) ima parametre, kroz **call**(*this_reference*, *args*) argumenti se proslijede standardno, dok se sa **apply**(*this_reference*, [*args*]) proslijede kao niz
- Glavna uloga funkcija call() i apply() je promjena reference this objekta
- Vidjećemo kako se ovo koristi kod nasleđivanja

call(), apply(), primjeri

```
var person = {
    name: "Tom",
    age: 18,
    getName: function(){
        return this.name
    }
}

var logInterests = function(interest1, interest2){
    console.log(this.getName()+" likes to play " + interest1 + " and " + interest2)
}

logInterests.call(person, "basketball", "baseball")    // "Tom likes to play basketball and basebal
logInterests.call(person, ["basketball", "baseball"])    // "Tom likes to play basketball and baseb
```

bind()

- Metod bind() kombinuje objekat sa funkcijom i vraće novu funkciju

```
var person = {
    name: "Tom",
    age: 18,
    getName: function(){
        return this.name
    }
}

var logName = function(){
    console.log("Logged: " + this.getName())
}

var logPersonName = logName()
logPersonName()    // "TypeError: this.getName is not a function

var logPersonName2 = logName().bind(person)
logPersonName2()    // "Logged: Tom"
```

Šta je ovdje **this**?

- Sa bind, **this** unutar funkcije se odnosi na objekat, npr. **this** unutar *logName* će se odnositi na person objekat

```
var logName = function(){
    console.log("Logged: " + this.getName())
}.bind(person)

logName()    // "Logged: Tom"
```

Pravila za this, bitno kako se funkcija poziva

- Unutar metode, **this** referencira ka objektu iz kog je pozvan metod (owner object) - **implicit binding**
- Metodi kao što su **call**(), **apply**() i **bind**() mogu da definišu referencu this objekta ka bilo kom objektu **- explicit binding**
- Unutar funkcije ili unutar Global Execution Context, **this** referencira ka globalnom objektu (u pretraživaču to je window objekat)
    - Unutar funkcije, strict mode, this je undefined
- Unutar događaja, **this** referencira ka elementu koji je prihvatio događaj
- Kod arrow funkcija **this** referencira ka istom objektu kao i metod (funkcija) u kome je arrow funkcija definisana

Pair Programming

- Raditi do Extension: Subclassing
  - http://csbin.io/oop

Resursi

- JavaScript: OOP (Frontend-Masters)
- JavaScript The Weird Parts (pogledajte additional material)
- JavaScript: Foundations (pogledajte additional material)
- https://linuk.gitbooks.io/notes-of-javascript-understanding-the-weird-parts/content/call-apply-and-bind.html

Pitanja