JavaScript

Sedamnaesti dio

Pregled

- Iteratori
- Generatori
- Pair Programming

Obnavljanje

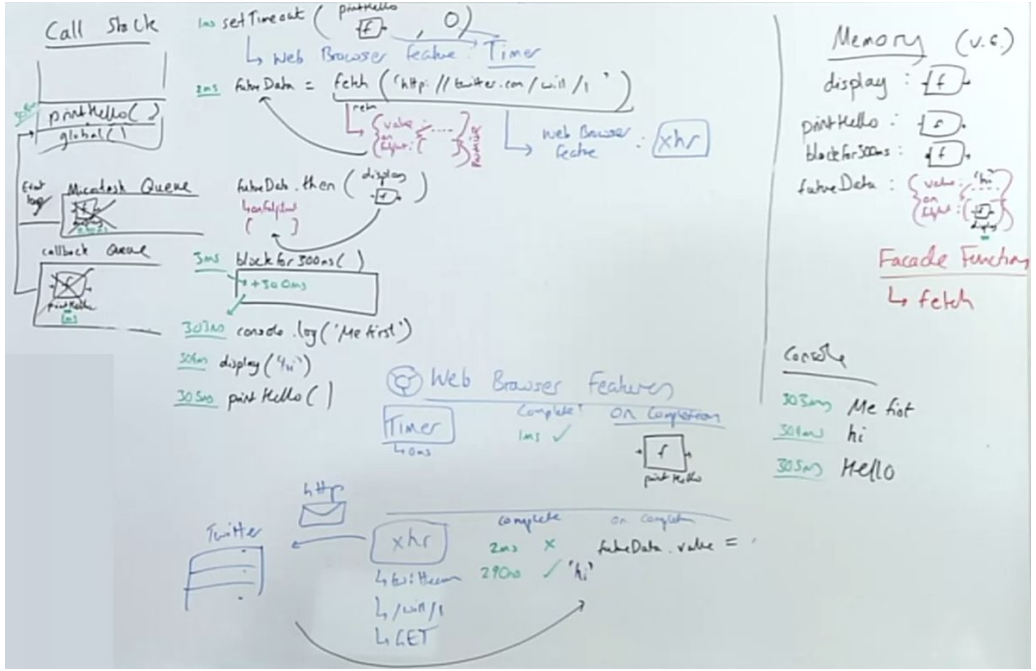
```
class userCreator{
  constructor  (name, score){
    this.name = name;
    this.score = score;
  }
  sayName (){
    console.log("I am " + this.name);
  }
  increment (){
    this.score++;
  }
}

const user1 = new userCreator("Phil", 4);
const user2 = new userCreator("Tim", 4);

user1.sayName()
```

```
class paidUserCreator extends userCreator {
  constructor(paidName, paidScore, accountBalance){
    super (paidName, paidScore);
    this.accountBalance = accountBalance;
  }
  increaseBalance (){
    this.accountBalance++;
  }
}

const paidUser1 = new paidUserCreator("Alyssa", 8, 25);

paidUser1.increaseBalance();

paidUser1.sayName();
```

Obnavljanje, drugi dio

```
function display(data){console.log(data)}
function printHello(){console.log("Hello");}
function blockFor300ms(){/* blocks js thread for 300ms with long for loop */}

setTimeout(printHello, 0);

const futureData = fetch('https://twitter.com/will/tweets/1')
futureData.then(display)

blockFor300ms()

// Which will run first?

console.log("Me first!");
```

Skica za prethodni primjer



Iteratori, motivacija za uvođenje

- Šta smo rekli za Promise, čemu služe, gdje se koriste, koje metode posjeduje? Sjećate li se šta je MicroTask queue? Čemu služi fetch()
- Vrlo često se u praksi srijećemo sa nekom listom ili kolekcijom elemenata gdje želimo da prođemo kroz sve elemente liste/kolekcije i da eventualno uradimo nešto sa tim elementima

```
const numbers = [4,5,6]

for (let i = 0; i < numbers.length; i++){
  console.log(numbers[i])
}
```

Može bolje nego prethodni način

- Da bismo primijenili funkciju nad kolekcijom podataka, ako koristimo prethodni primjer, moramo da odradimo dvije stvari
  - Da pristupimo elementu
  - Da izvršimo funkciju nad elementom
- Iteratori automatizuju proces pristupanja svakom elementu, tako da samo trebamo da se fokusiramo na to šta treba da se odraditi sa konkretnim elementom
- Zamislite da možemo da kreiramo funkciju koja čuva numbers (iz prethodnog primjera) i svaki put kad je pozovemo ona nam vrati sledeći element (svakako treba na neki način pamtiti prethodni element, da bi smo znali koji je sledeći
- Šta misliti šta nam u ovome može pomoći, a govorili smo o tome?

Podsjećanje na Higher-Order funkcije

```
function createNewFunction() {
  function add2 (num){
    return num+2;
  }
  return add2;
}

const newFunction = createNewFunction()

const result = newFunction(3)
```

Funkcija koja ispunjava sve naše zahtjeve

- Funkcija koja čuva niz, prati gdje se iterator u kolekciji trenutno nalazi, svaki put kad je pozovemo vraće sledeći element.

```
function createFunction(array){
  let i = 0
  function inner(){
    const element = array[i];
    i++;
    return element;
  }
  return inner
}

const returnNextElement = createFunction([4,5,6])
const element1 = returnNextElement()
const element2 = returnNextElement()
```

Iterator funkcija

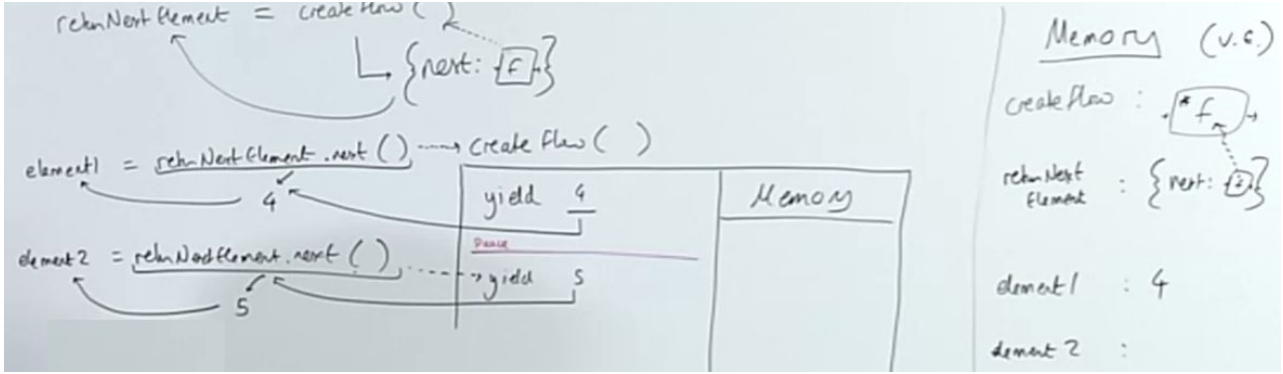
```
function createFlow(array){
  let i = 0
  const inner = {next :
      function(){
        const element = array[i]
        i++
        return element
      }
  }
  return inner
}

const returnNextElement = createFlow([4,5,6])
const element1 = returnNextElement.next()
const element2 = returnNextElement.next()
```

Generator funkcija, uvod

- Kako da podatke čuvamo u funkciji kao segmente, a ne u jednu kolekciju, a kako onda da im pristupamo pojedinačno?

```
function *createFlow(){
  yield 4
  yield 5
  yield 6
}

const returnNextElement = createFlow()
const element1 = returnNextElement.next()
const element2 = returnNextElement.next()
```

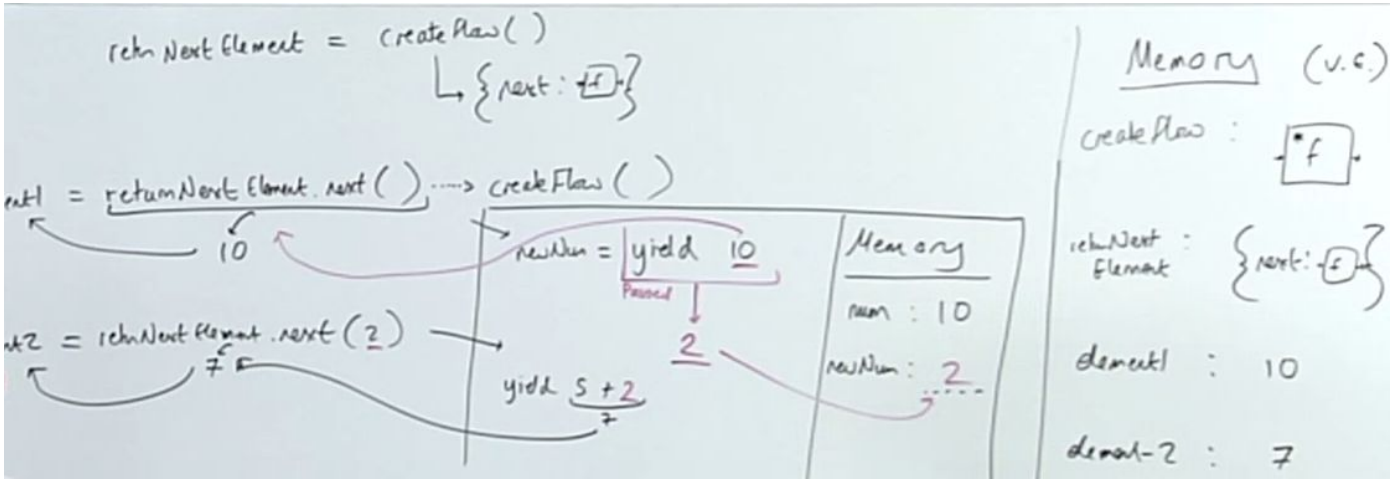
Skica


{next: f}
element1 = returnNextElement.next() → createFlow()
4
yield 4
Memory
element2 = returnNextElement.next()
yield 5
5
Memory (v.c.)
createFlow : f
returnNext Element : {next: f}
element1 : 4
element2 :

Generator funkcija, primjer 2

- returnNextElement specijalni objekat (generator objekat) koji kada se pozove next() metod nad njim pokreće createFlow() sve dok ne dođe do yield i vrati yielded vrijednost

```
function *createFlow(){
  const num = 10
  const newNum = yield num
  yield 5 + newNum
  yield 6
}

const returnNextElement = createFlow()
const element1 = returnNextElement.next() // 10
const element2 = returnNextElement.next(2) // 7
```

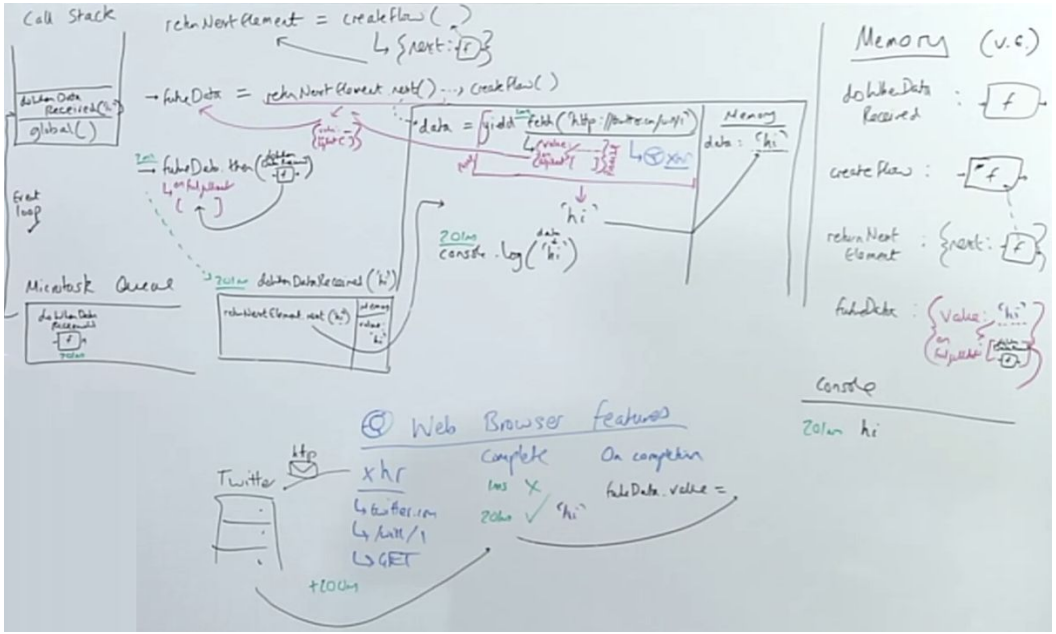

returnNextElement = createFlow()
{next: f}
= returnNextElement.next() ----> createFlow()
10
newNum = yield 10
Paused
2
= returnNextElement.next(2)
7
yield 5 + 2
7
Memory
num : 10
newNum : 2
Memory (v.c.)
createFlow : f
returnNext Element : {next: f}
element1 : 10
element2 : 7

Asinhroni generatori

- Kod asinhronosti u JSu želimo:
  - Da na neki način pokrenemo task koji traje duže (npr. slanje zahtjeva ka serveru)
  - Da sinhroni kod u međuvremenu nastavi da se izvršava
  - Da pokrenemo neku funkionalnost kad nam npr. stignu podaci sa server

```
function doWhenDataReceived (value){
  returnNextElement.next(value)
}

function* createFlow(){
  const data = yield fetch('http://twitter.com/will/tweets/1')
  console.log(data)
}

const returnNextElement = createFlow()
const futureData = returnNextElement.next()

futureData.then(doWhenDataReceived)
```

Skica za prethodni primjer



Asinhrone funkcije

- FUnkcija koja sadrži await keyword mora da sadrži ispred function **async** keyword
- **await** keyword se stavlja ispred funkcija koje vraćaju promise objekat

```
async function createFlow(){
  console.log("Me first")
  const data = await fetch('https://twitter.com/will/tweets/1')
  console.log(data)
}

createFlow()

console.log("Me second")
```

for...of

- Poziva returnNextElement() sve dok ima elemenata u kolekciji
  - Automatski kreira returnNextElement() funkciju
  - Poziva returnNextElement() funkciju i čuva vraćeni element u element koji se dalje koristi kroz petlju

```
const numbers = [4,5,6]

for (let element of numbers){
  console.log(element)
}
```

Pair Programming

- http://csbin.io/promises
- http://csbin.io/iterators

Reference

- JavaScript: The New Hard-Parts (Frontend-Masters)

Pitanja