wantsome
The friendly IT Academy

Saptamana 12

Partea 1

Programare Front-End

1. Functional Programming

What is functional programming ?

Programarea functionala este:

- o **paradigma de programare**

- procesul prin care putem scrie cod cu ajutorul **pure functions** - astfel se evita ***shared state***, ***mutable data*** si ***side-efects***

- **declarativa** si NU imperativa

Concepte de baza in programarea functionala

- **Pure function**
- **Function composition**
- **Avoiding shared state**
- **Avoiding mutating state (Mutability / Immutability)**
- **Avoiding side efects**

1.1 Functional programming core concepts

Pure function

O *functie pura* este o functie pentru care valoarea returnata este determinata doar de valorile de *input,* fara producerea altor efecte secundare (pentru aceleasi valori de *input,* rezultatul va fi mereu acelasi ori de cate ori se apeleaza functia pura)

Exemple:

- Math.random() - **functie impura** - *rezultatul va fi diferit de fiecare data*
- Math.min(1,2) - **functie pura** - *rezultatul va fi mereu acelasi*

Pure function

```
function add2 (x){
  return x + 2
}
```

vs.

```
var y = 2
function adder (x){
  return x + y
}
```

Shared state

- orice variabila, obiect sau spatiu de memorie care exista intr-un *scope* comun sau care imparte proprietati intre scope-uri

- un *scope* comun sau *shared* poate fi *scope*-ul global sau *closure scope*

- de obicei, in *OOP*, obiectele se distribuie intre *scope*-uri prin adaugarea de proprietati la alte obiecte

Problema cu impartirea (sharing) *state*-ului este faptul ca trebuie sa fii constient si sa intelegi de unde provine intreaga istorie a unei variabile pentru a intelege efectele schimbarii valorilor ei.

Programarea functioanala evita shared state-ul prin derivarea de date noi din datele existente!
(*derived state*)

Function composition

```
// With shared state, the order in which function calls are made
// changes the result of the function calls.
const x = {
  val: 2
};
const x1 = () => x.val += 1;
const x2 = () => x.val *= 2;

x1();
x2();
console.log(x.val); // 6
```

```
const y = {
  val: 2
};
const y1 = () => y.val += 1;
const y2 = () => y.val *= 2;
// ...the order of the function calls is reversed...
y2();
y1();
// ... which changes the resulting value:
console.log(y.val); // 5
```

```
const x = {
  val: 2
};
const x1 = x => Object.assign({}, x, { val: x.val + 1});
const x2 = x => Object.assign({}, x, { val: x.val * 2});
console.log(x1(x2(x)).val); // 5
```

```
const y = {
  val: 2
};
x2(y);
x1(y);
console.log(x1(x2(y)).val); // 5
```

[Function composition]

Immutability

- Un obiect imutabil (*immutable object*) este un obiect ce nu poate fi modificat dupa momentul in care acesta a fost creat

- Imutabilitatea este conceptul central al programarii functionale pentru ca astfel se evita *state history*, practic exista siguranta legata de faptul ca valorile unei variabile nu se vor schimba odata cu executarea unei functii sau alteia

- Nu asociati ***const*** cu imutabilitatea - **CONST** creeaza variabile ce nu pot fi reasignate, nu creeaza obiecte imutabile

  - Nu putem schimba obiectul la care se refera dar putem schimba sau adauga proprietatile obiectului ceea ce face ca acesta sa fie ***mutabil***

  Se pot crea obiecte *immutable* folosind metoda **Object.freeze**(), insa doar pentru **primul nivel**

Immutability

Immutability & Object.freeze()

```
const a = Object.freeze({
  foo: 'Hello',
  bar: 'world',
  baz: '!'
});
a.foo = 'Goodbye';
// Error: Cannot assign to read only property 'foo' of object Object
```

```
const a = Object.freeze({
  foo: { greeting: 'Hello' },
  bar: 'world',
  baz: '!'
});
a.foo.greeting = 'Goodbye';
console.log(`${ a.foo.greeting }, ${ a.bar }${a.baz}`);
```

Immutability

Side effects

- orice schimbare a starii unei aplicatii care este observabila in afara functiei care produce acest efect

- rezultat care afecteaza nu doar comportamentul intern al unei functii ci al tuturor celorlalte functii care folosesc datele afectate

- modificarea oricarei variabile externe sau proprietate de obiect

- Atunci cand vorbim despre **programare functionala,** ne rezumam foarte des la folosirea conceptului de a tine separat *side effects* de restul logicii al aplicatiei - astfel, codul va fi mult mai usor inteligibil si mult mai usor de depanat (*debugging*) pentru viitor

side-effects

Higher Order Functions (**HOF**s)

O functie de tip **HOF** este o functie care indeplineste cel putin una dintre urmatoarele conditii:

- primeste ca argument una sau mai multe functii
- returneaza ca si rezultat o functie

- sunt folosite la abstractizarea sau izolarea actiunilor, efectelor sau *flow*-ului de callbacks.

side-effects

HOF - three most used functions

Filter - avem un array si vrem doar anumite elemente din el.

- http://bit.do/HOFfilter

Map - avem un array de obiecte si vrem doar id-urile din el

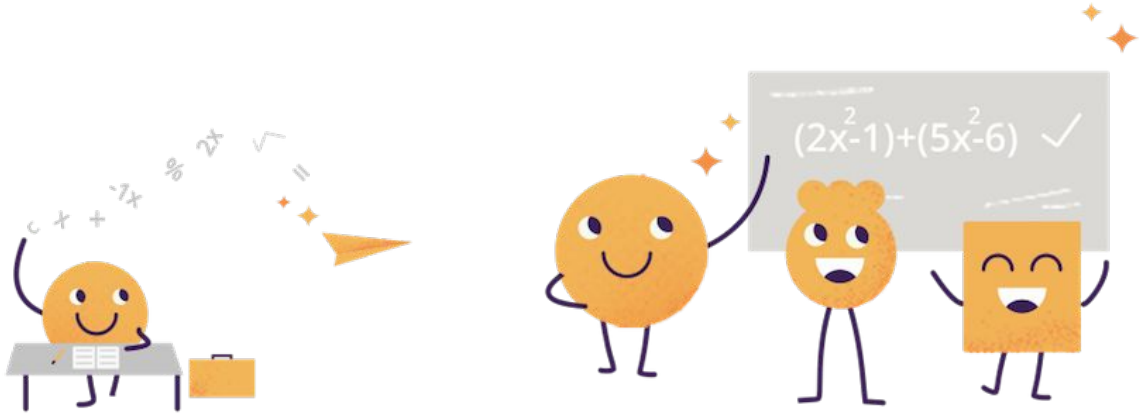
- http://bit.do/HOFmap

Reduce - http://bit.do/HOFreduce

Filter, Map, Reduce explained very well

Filter() map() reduce()

PRACTICE: **Functional Programming in JS**

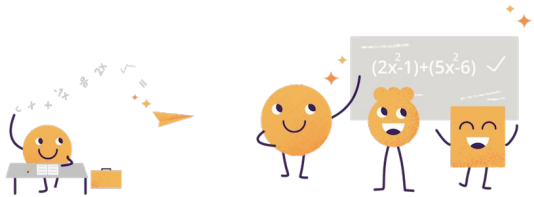


PRACTICE: **Functional Programming in JS**

Cerinte:

1.

```
function capitalizeNames(arr){
  // your code here
}

console.log(capitalizeNames(["john", "JACOB", "jinGleHeimer", "schmidt"]));

// ["John", "Jacob", "Jingleheimer", "Schmidt"]
```

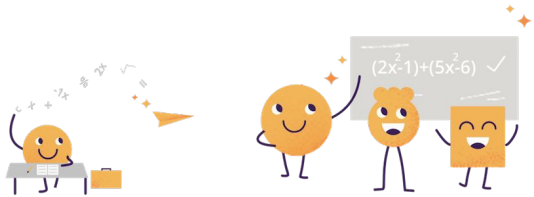


PRACTICE: **Functional Programming in JS**

Cerinte:

2. Scrieti codul corespunzator functiei care dubleaza fiecare element de tip valoare numerica dintr-un array specificat ca argument

```
function doubleEachNumber(arr){
  // write your code in here
}

console.log(doubleEachNumber([2, “5”, 100, “100”, “blalblala”])); // [4, “5”, 200, “100”, “blalblala”]
```



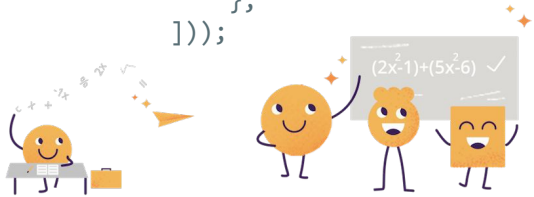
PRACTICE: **Functional Programming in JS**

Cerinte:

3.

```
function getPersonsNames(arr){
  // your code here
}

console.log(getPersonsNames([
    {
        name: "Angelina",
       surname: “Jolie”,
        age: 80
    },
    {
        name: "Eric",
       surname: “Jones”,
        age: 27
    },
]));
```

```
// ["Angelina Jolie", "Eric Jones"]
```



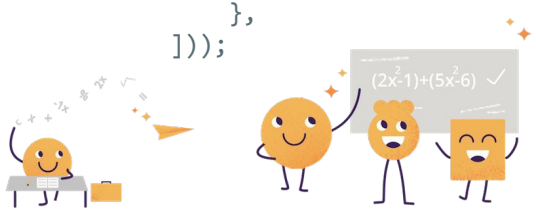
wantsome
The friendly IT Academy

PRACTICE: **Functional Programming in JS**

Cerinte:

4.
```
function computeExamPass(arr){
  // your code here
}

console.log(computeExamPass([
    {
        name: "Angelina",
       surname: “Jolie”,
        grade: 7
    },
    {
        name: "Eric",
       surname: “Jones”,
        grade: 3
    },
]));
```

```
[

"Angelina Jolie has passed the
exam",

"Eric Jones has not passed the exam"

]
```



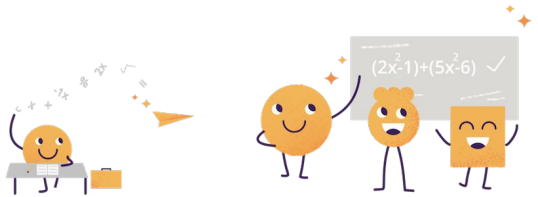
PRACTICE: **Functional Programming in JS**

Cerinte:

5. Implementati functia de mai jos si utilizati rezultatul pentru a afisa elementele intr-o pagina

```
function getPersonsDomElements(arr){
  // your code here
}

console.log(getPersonsDomElements([
    {
        name: "Angelina",
       surname: “Jolie”,
        age: 80
    },
    {
        name: "Eric",
       surname: “Jones”,
        age: 27
    },
]));
```

```
// [

"<h1>Angelina Jolie"<h1><h2>80</h2>”,

“<h1>Eric Jones</h1><h2>27</h2>”

]
```



wantsome
The friendly IT Academy

PRACTICE: **Functional Programming in JS**

http://bit.do/ex1FP
http://bit.do/ex2FP
http://bit.do/FPex3
http://bit.do/FPex4
http://bit.do/FPbonus1
http://bit.do/FPbonusHard