Programação Funcional

Programação para Internet I

IFB - Tecnologia em Sistemas para Internet

Marx Gomes van der Linden

Funções de primeira classe

- Uma função também é um objeto
- Pode ser armazenada em uma variável, passada e retornada em outras funções

```
function ola(){
     return "Olá, mundo!";
}

let x = ola();
let y = ola;
let z = y();
```

Funções anônimas

- Uma função de primeira classe não precisa ter nome

```
let z = function(a, b) {
      return a + b
};
let x = z;
let resultado = x(5, z(3,4));
```

Funções como parâmetros

```
let soma = function (a, b) {
      return a + b
};

function operacao(op, x, y){
    return op(x, y);
}

let v1 = operacao(soma, 10, 5);
let v2 = operacao(
   function(a,b){return a * b}, v1, 4
);
console.log(v1, v2);
```

Sintaxe alternativa (ES2015)

```
let soma = (a, b) => a + b;



function operacao(op, x, y){
    return op(x, y);
}

let v1 = operacao(soma, 10, 5);
let v2 = operacao( (a, b) => a * b, v1, 4);


console.log(v1, v2);
```

Sintaxe alternativa (ES2015)

```
let f1 = function(a,b) { return a*b };
let f2 = (a,b) => { return a*b };
let f3 = (a,b) => a*b;

console.log(f1(3,4), f2(3,4), f3(3,4));

let f4 = function(a) { return a*a };
let f5 = (a) => { return a*a };
let f6 = (a) => a*a;
let f7 = a => a*a;

console.log(f4(5), f5(5), f6(5), f7(5));
```

Funções em outras estruturuas

```
let a = () => { console.log('A!') };

let b = [
    () => { console.log('B zero') },
    () => { console.log('B um') }
];

let c = () => {
    return () => {  console.log("C?")  }
};

let d = {
    primeiro: () => { console.log("D1") },
    segundo: () => { console.log("D2") }
};
```

Funções em outras estruturas

```
a();
b[0]();
b[1]();
c()();
d.primeiro();
d.segundo();
```

Array: método **forEach**

- O método **forEach** aplica uma função a cada elemento do array

```
const numeros = [4, 10, 6, 7, 5];
let soma = 0;
numeros.forEach( function(x) { soma += x } );
console.log(soma);
```

Array: método `filter`

- O método **`filter`** recebe uma função booleana e retorna um array contendo os elementos ”aprovados” por aquela função

```
function ehPar(x){
    return x % 2 == 0;
}

const numeros = [12,324,213,4,2,3,45,4234];
const pares = numeros.filter(ehPar);
console.log(pares);
```

Array: método **map**

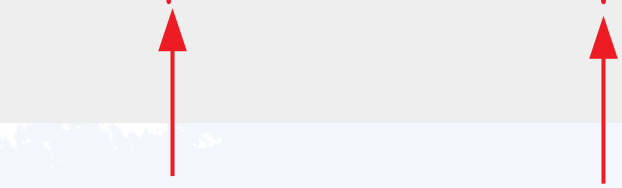
- O método **map** aplica uma função a cada elemento do array e substitui o valor no array pelo valor retornado pela função

```
const numeros = [1,2,3,4,5,6,10,20];
const quadrados = numeros.map( x => x*x );
console.log(quadrados);
```

Array: método **reduce**

- Aplica uma função com dois argumentos:
  - » (*valor_acumulado*, *próximo_valor*)
- Aplica a todos os elementos, passando cada retorno como o valor acumulado da próxima chamada

```
const a = [1,2,3,4,5];
let soma = a.reduce((ac,x) => ac+x , 0);
console.log(soma);
```



Array: método **reduce**

- O acumulador pode ser um array

```
let minhafuncao = (ac, x) => {
    if(x % 2 == 0)
        ac.push(x);

    return ac;
};

const numeros = [12,324,213,4,2,3,45,4234];
let pares = numeros.reduce(minhafuncao, []);

console.log(pares);
```

Array: método **sort**

- Ordena um array, a princípio em ordem alfabética
- Se for passada uma função com dois parâmetros, usa essa função para comparar os elementos
- Retorno da função (**a**,**b**) fornecida:
  - **<0** → **a** deve ser considerado *menor* que **b**
  - **>0** → **a** deve ser considerado *maior* que **b**
  - **==0** → **a** e **b** são *iguais* para efeitos de ordenamento

Array: método **sort**

```
let lista = [ 4, 10, 3, 20, 30 ];
lista.sort();
console.log(lista);

lista.sort( (a,b) => a - b );
console.log(lista);
```

Função comparadora

Função que retorna uma função

- Uma função pode retornar uma função

```
function funcaoop(operador){
    if(operador == '+') return (a, b) => a + b;

    if(operador == '-') return (a, b) => a - b;

    if(operador == '^') return (a, b) => a ** b;
}

let somador = funcaoop('+');
let potencia = funcaoop('^');

console.log(somador(5,6), potencia(2,5));
console.log( funcaoop('-')(20,5) );
```

Closures

- **Closure** é o contexto interno em que uma função é criada, que é incorporado em seu retorno

```
function fazfuncao(){
    let texto = "Olá, mundo!";

    return function() {
        return texto;
    }
}

let func = fazfuncao();
console.log(func());
```

Closures

```
function fazfuncao(saudacao){
    return function(quem) {
        return `${saudacao}, ${quem}!`;
    }
}

let func1 = fazfuncao("Olá");
let func2 = fazfuncao("Hello");
let func3 = fazfuncao("Bonjour");

console.log(func1('José'));
console.log(func2('Antônio'));
console.log(func3('Maria'));
```

Immediately-invoked Function Expressions (IIFE)

- Expressões de função invocadas imediatamente
- Uma função anônima é definida e, logo em seguida, invocada
  - » Serve para modularizar o código
  - » Vem sendo substituído por módulos

Immediately-invoked Function Expressions (IIFE)

```
(function() {
        function ola(){
            console.log('Olá, Mundo!');

        }

        ola();

    }
)();

ola(); ← Erro!
```