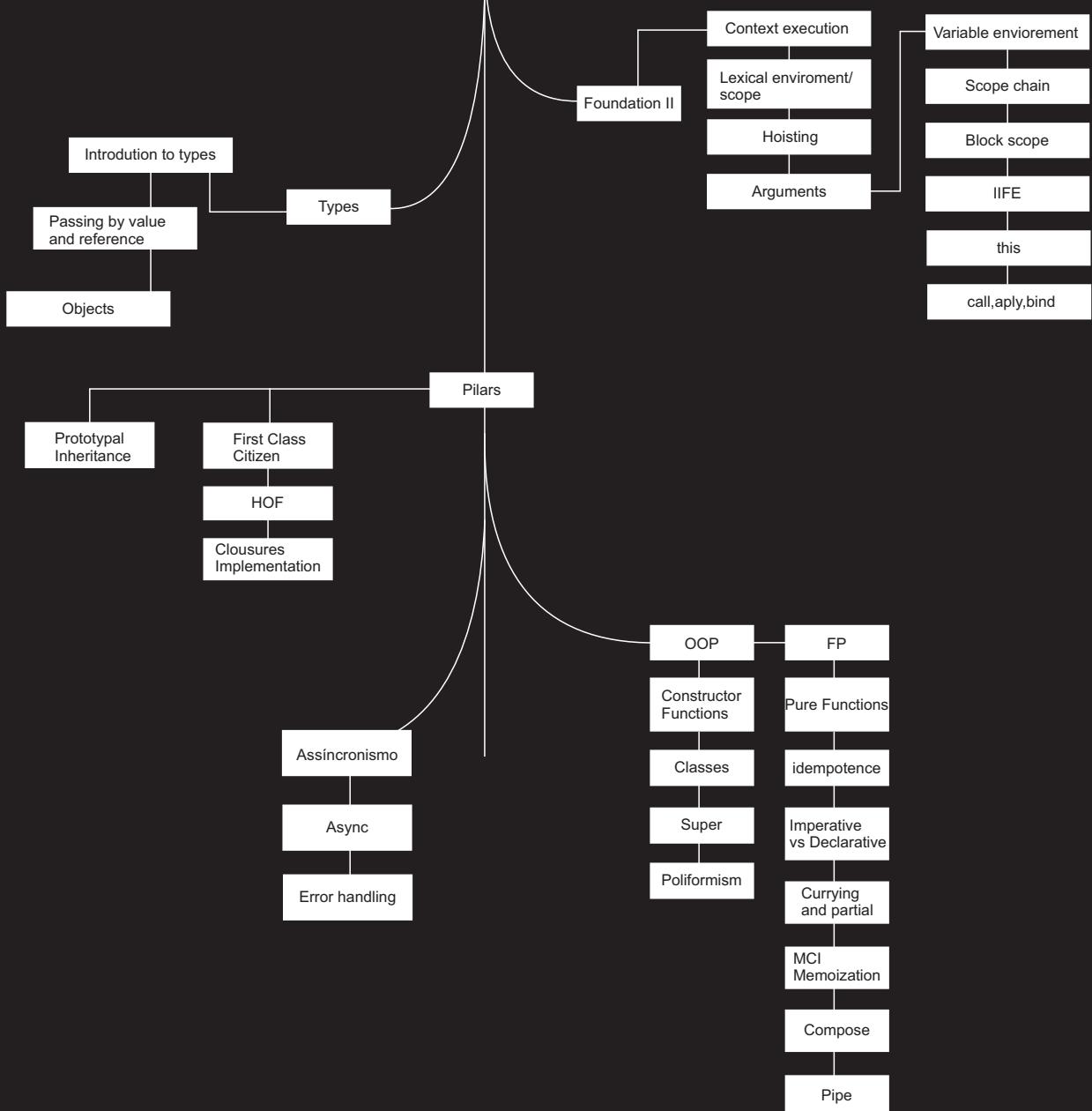


# JS



# CONTEXT EXECUTION

No Javascript, cada função é compreendida como um contexto de execução (context execution), e absolutamente todos estes contextos são acoplados para dentro de um contexto global de execução

```
function myName(){
    return 'Gabriel Cerqueira'
}

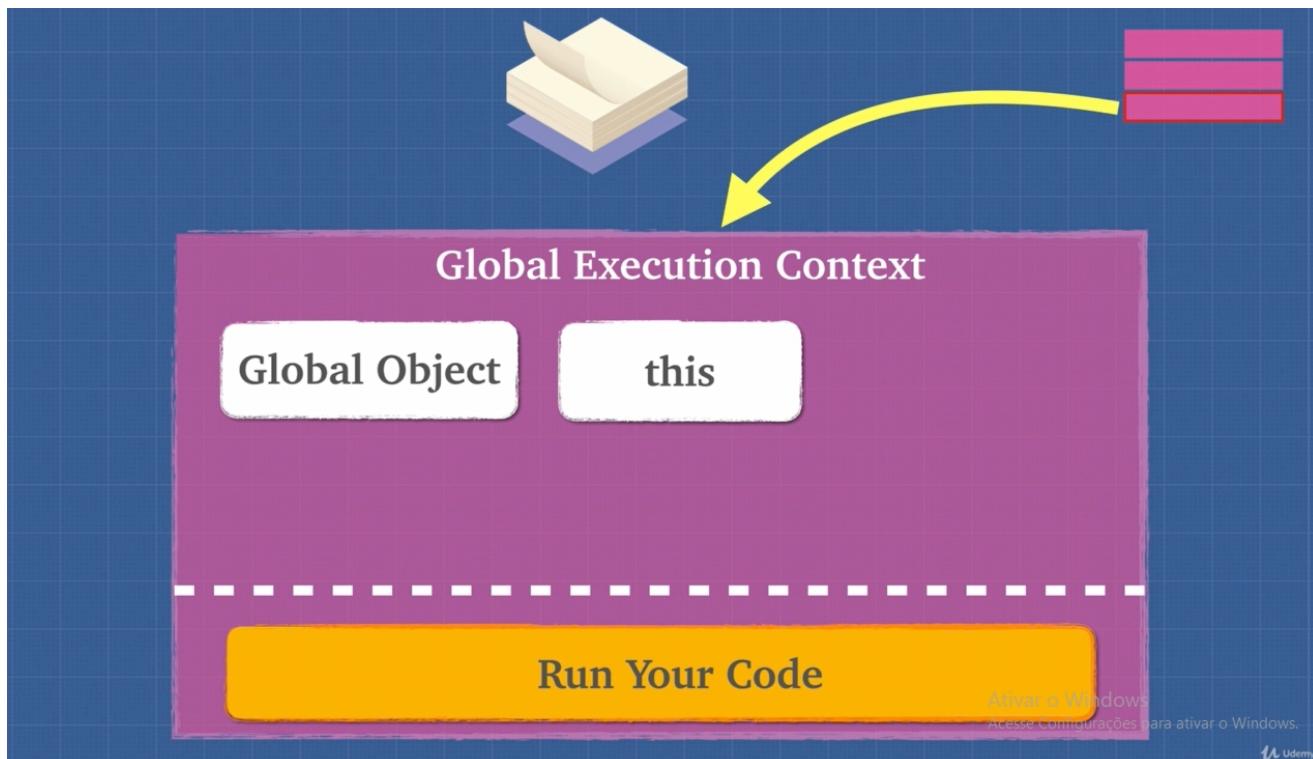
function findName(){
    return myName();
}

function sayName(){
    findName();
}

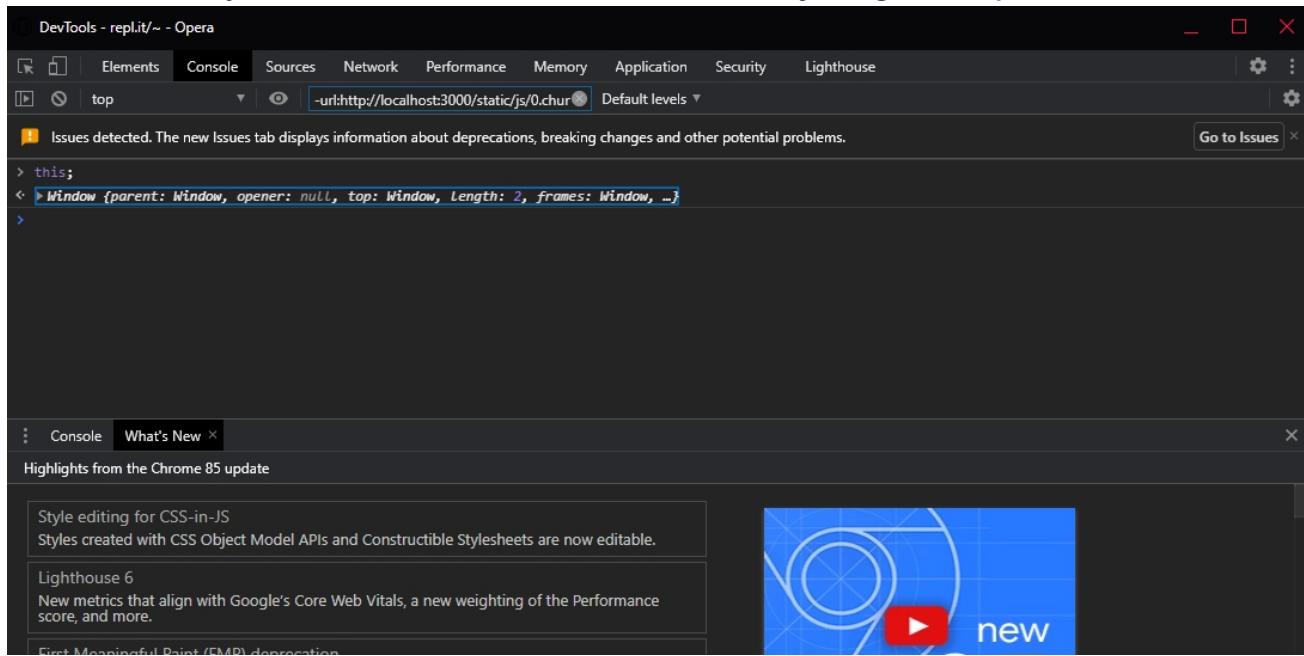
sayName();

//GlobalContext()
//sayName()
//findName()
//myName()
```

Internamente a este objeto global é atribuído duas informações, um objeto(global) e a palavra chave 'this'.



Para exemplo, podemos abrir o navegador na aba de console presente na inspeção e acessar a key word ‘this’. Nos será exibido o objeto global que foi mencionado



The screenshot shows the Opera DevTools interface with the 'Console' tab selected. The URL bar indicates the page is 'localhost:3000/static/js/0.chunk.js'. In the console, the command '`> this;`' is entered, and the result is displayed as '`<Window {parent: Window, opener: null, top: Window, length: 2, frames: Window, ...}`'. Below the console, a 'What's New' panel is open, showing highlights from the Chrome 85 update, including 'Style editing for CSS-in-JS' and 'Lighthouse 6'. A large blue button with a play icon and the word 'new' is visible.

através dele nos é fornecido diversas funções como exemplo, setTimeOut, setInterval (providas através de web api's);

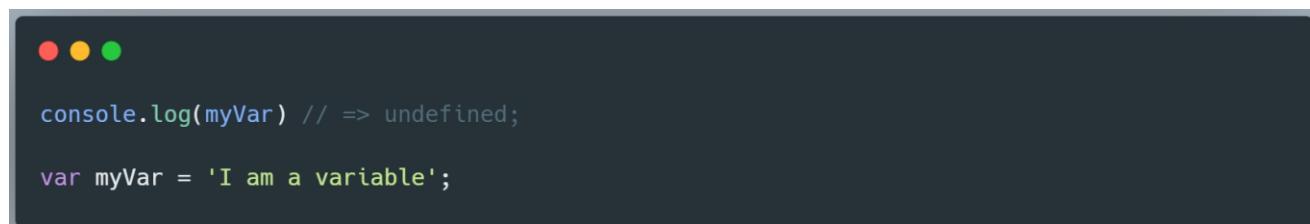
## LEXICAL ENVIRONMENT/SCOPE

Lexical Environment é um simples mecanismo usado no processo de compilação para o Javascript identificar onde ‘determinado’ trecho de código está sendo escrito. Já o Lexical Scope, determina onde as variáveis/dados estão sendo inseridos e a quem vós pertence.

## HOISTING

Conceito base do hoisting: elevar dados a um escopo superior para que seja acessível a todo contexto léxico interno.

Exemplo 1:



```
● ● ●
console.log(myVar) // => undefined;
var myVar = 'I am a variable';
```

estou acessando minha variável ‘myVar’ antes mesmo de sua declaração e não é me retornando nenhum erro de referência. Porém seu valor só é de fato inicializado corretamente de acordo com o fluxo lógico do algoritimo.

## Exemplo 2:

```
● ● ●  
console.log(a()); // => A2  
  
function a(){  
  console.log('A1');  
}  
  
function a(){  
  console.log('A2');  
}
```

podemos declarar dados com o mesmo nome, porém a leitura do escopo local sempre irá considerar o último definido.

## Exemplo 3:

```
● ● ●  
var food = 'Banana';  
  
function myFavouriteFood(){  
  console.log("My favorite food is", food);  
  
  var food = "Sushi";  
  
  console.log("My Second favorite food is", food);  
}  
  
myFavouriteFood() //=> My favorite food is undefined  
  
                  //=> My Second favorite food is Sushi;
```

Uma variável definida no escopo local sempre será prioridade, por esse motivo o primeiro console buscou o valor de ‘food’ internamente e já que a atribuição só é feita em um fluxo lógico como já mencionado anteriormente, o valor foi exibido como indefinido.

# ARGUMENTS

No Javascript temos os argumentos esperados de funções, eles são inseridos para dentro de um objeto acessível internamente no contexto único com a palavra chave ‘arguments’

```
● ● ●  
  
function cars(car1, car2){  
  console.log('arguments', arguments);  
  return `car ${car1} is better ${car2}`  
};  
  
cars('fiat', 'ford') // () =>  
//arguments [Arguments] { '0': 'fiat', '1': 'ford' }  
//car fiat is better ford;
```

para evitar problemas de acesso podemos converter os argumentos em um array usando as seguintes formas.

Exemplo 1: Com o método ‘Array.from’

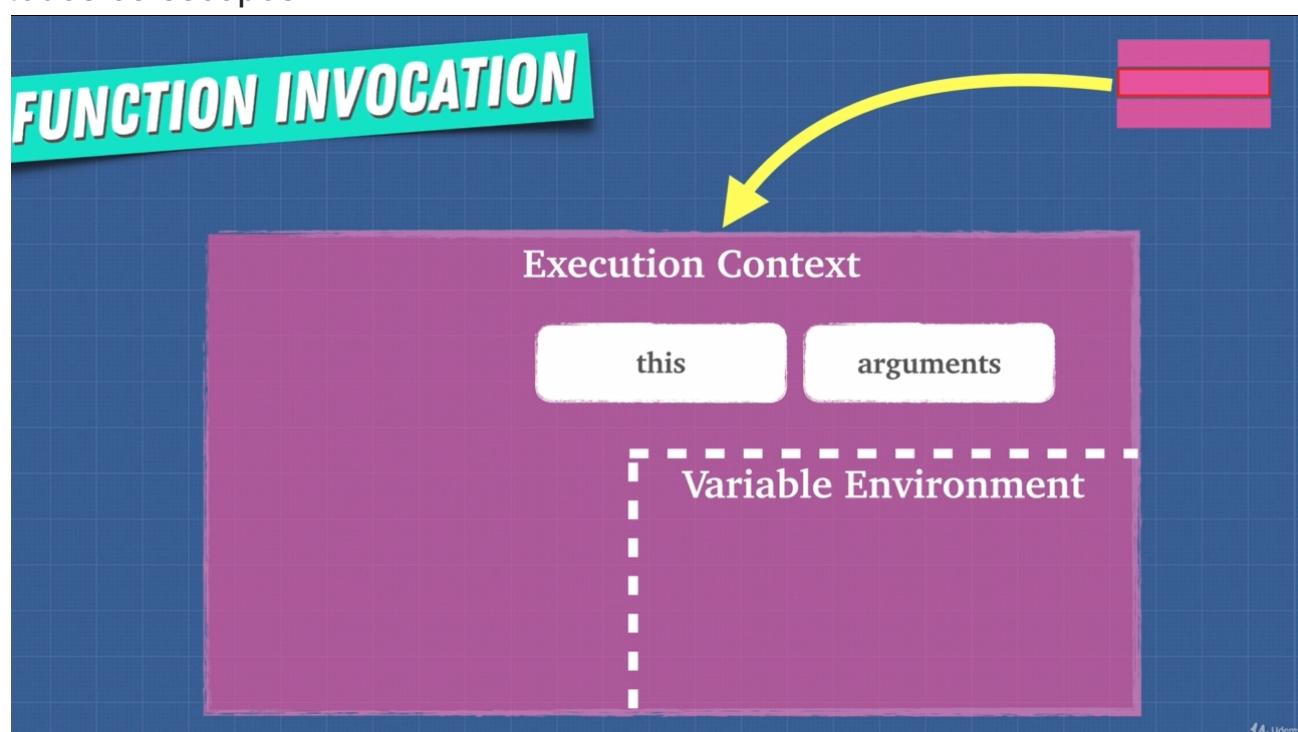
```
function cars(car1, car2){  
  console.log(Array.from(arguments));  
  return `car ${car1} is better ${car2}`  
};  
  
cars('fiat', 'ford') // () =>  
// [ 'fiat', 'ford' ]  
// car fiat is better ford
```

Exemplo 2: Com o spread operator

```
function cars(...args){  
  console.log(args);  
  return `car ${args[0]} is better ${args[1]}`  
};  
  
cars('fiat', 'ford') // () =>  
// [ 'fiat', 'ford' ]  
// car fiat is better ford
```

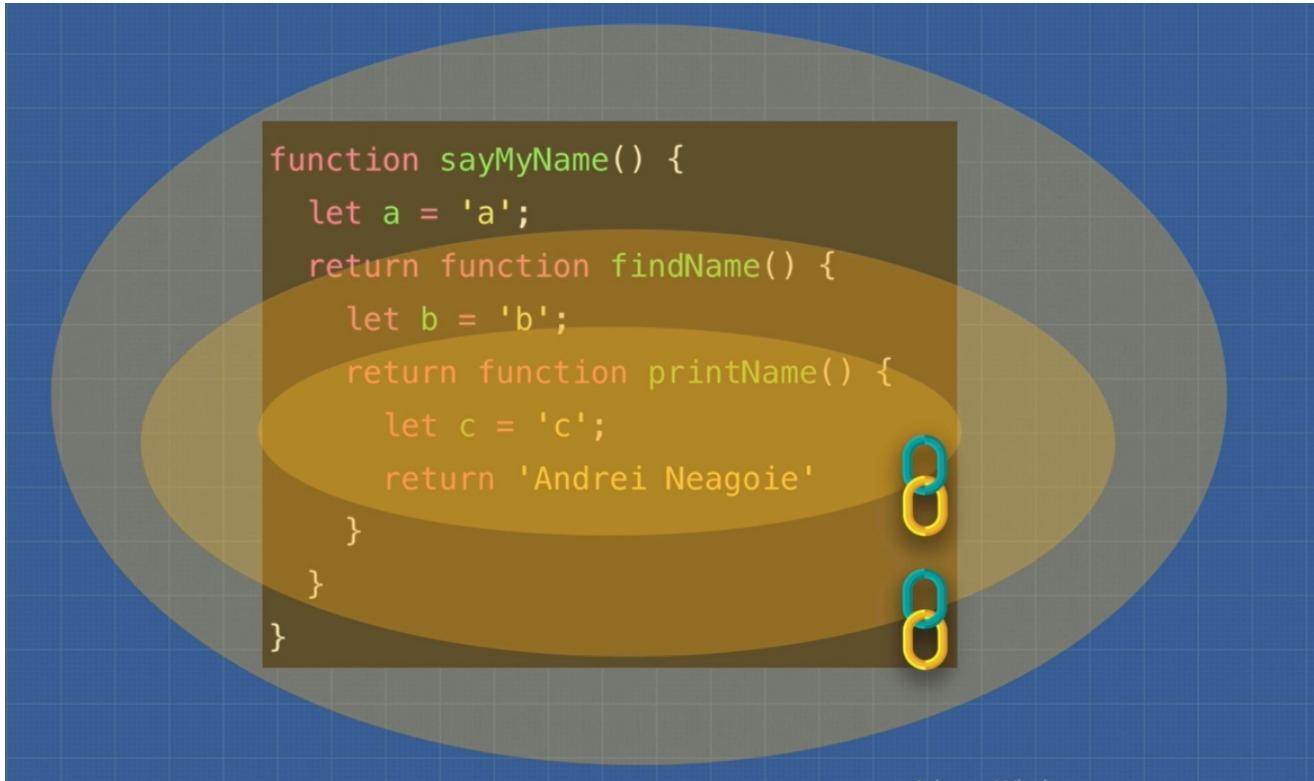
## VARIABLE ENVIRONMENT

Cada contexto de execução além de receber argumentos, a key word ‘this’, também recebe um espaço reservado para alocação de memória em seu ambiente interno a todos os escopos.



# SCOPE CHAIN

Ok, falamos um pouco anteriormente a respeito de contexto de execução, entendemos que todo contexto detém o seu próprio ambiente de variáveis, e além disso um: escopo único que é conectado com os demais escopos em uma cadeia de pai e filhos.



Os escopos de alguma forma estão conectados, então ao tentar acessar um valor ‘inexistente’ no ambiente de variável do escopo atual, a engine buscará nos escopos ‘fathers’ até que seja encontrado.



```
function scopeA(){  
  var varA = 'A';  
  
  function scopeB(){  
    function scopeC(){  
      console.log(varA);  
    }  
  
    scopeC();  
  }  
  
  scopeB();  
}  
  
scopeA() // => A;
```

# BLOCK SCOPE

Escopos de blocos são definidos dentro de condicionais e loop's, e diferentemente do escopo padrão de funções, os seus valores podem ser acessados fora do bloco com a definição do VAR.

```
● ● ●  
if(5 > 4){  
  var myVar = true;  
}  
  
console.log(myVar) //=> true;
```

```
● ● ●  
for(var i = 0; i < 1; i++){  
  var x = 2;  
}  
  
console.log(i, x) // => 1, 2;
```

# IIFE

IIFE é uma forma de declarar funções de expressão que são imediatamente invocadas

```
● ● ●  
(function(){  
  console.log('I am invoked')  
})() // => I am invoked;
```

O conteúdo presente nos primeiros parênteses é a construção da função ‘anônima’ e os parênteses seguintes indicam a invocação da mesma.

Também podemos guardar essa declaração em memória e criar funções internas acessíveis

```
● ● ●  
var script = (function(){  
  function sayName(name){  
    console.log(name);  
  }  
  
  return {  
    sayName: sayName  
  }  
})();  
  
script.sayName('My name is gaba');
```

# THIS

A keyword ‘this’ serve para apontar valores na qual a função é uma propriedade, isto na prática fica da seguinte forma:

```
● ● ●  
const obj = {  
  a: 'I am A,',  
  b: 'I am B',  
  
  say( ){  
    console.log(this.a, this.b);  
  }  
}  
  
obj.say() // => I am A, I am B;
```

Ao invocarmos a função ‘say’ presente no objeto e usar o ‘this’ para apontar para os valores de ‘a’ e ‘b’, foi me retornado as propriedades do objeto qual a função está presente

Outro exemplo:

```
● ● ●  
function importantPerson( ){  
  console.log(this.name)  
};  
  
var name = 'Gaba';  
  
importantPerson();
```

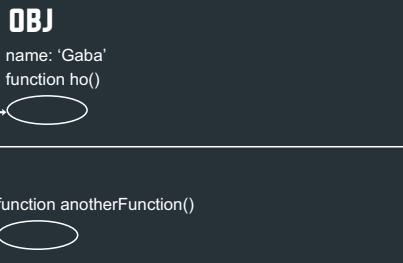
A função ‘importantPerson’ está presente em um ‘objeto global’, assim como a propriedade ‘name’ e por este fator conseguimos extrair o valor do próprio name.

Aqui vamos com uma ‘pegadinha’.

```
● ● ●  
const obj = {  
  name: 'Gaba',  
  ho(){  
    console.log('a', this);  
  
    var anotherFunction = function( ){  
      console.log('b', this);  
    }  
  
    anotherFunction();  
  }  
}  
obj.ho();  
// ('a', obj: { name: 'Gaba' } )  
// ('b', globalObject: { ... } )
```

em vez da função ‘anotherFunction’ exibir o mesmo objeto invocado na primeira linha da função ‘ho’, foi retornado um objeto global. Isso acontece por fatores de escopos dinâmicos que são automaticamente assimilados, é algo que diz: “não importa onde a função foi escrita, e sim como”

## CONTEXT GLOBAL



e para que seja resolvido, basta declarar as funções utilizando ‘arrow functions’.



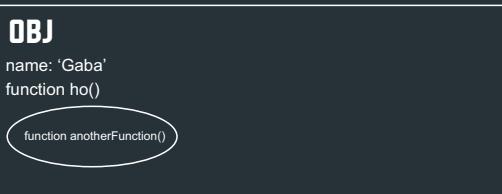
```
const obj = {
  name: 'Gaba',
  ho(){
    console.log('a', this);

    var anotherFunction = () => {
      console.log('b', this);
    }

    anotherFunction();
  }
}

obj.ho();
// ('a', obj: { name: 'Gaba' } )
// ('b', obj: { name: 'Gaba' } )
```

## CONTEXT GLOBAL



# CALL

A função principal do método ‘CALL’ é tomar emprestado uma função de um objeto, para que seja executado em um objeto passado.

```
● ● ●  
const people = {  
  name: 'Gabriel',  
  energy: 50,  
  setEnergy(n){  
    return this.energy += n;  
  }  
};  
const people2 = {  
  name: 'Gabriel',  
  energy: 50  
}  
people.setEnergy.apply(people2, 10);  
console.log(people2);  
// => { name: 'Gabriel', energy: 60 }
```

Sendo que o primeiro parâmetro da função ‘call’ é o objeto que queremos impor para dentro do método. Já os demais parâmetros serão enviados para a função que estamos manipulando.

# APPLY

A função ‘APPLY’ faz exatamente o mesmo trabalho do ‘call’, sendo a única diferença a passagem de parâmetros que é feita por arrays.

```
● ● ●  
const people = {  
  name: 'Gabriel',  
  energy: 50,  
  setEnergy(n){  
    return this.energy += n;  
  }  
};  
const people2 = {  
  name: 'Gabriel',  
  energy: 50  
}  
people.setEnergy.apply(people2, [10]);  
console.log(people2);  
// => { name: 'Gabriel', energy: 60 }
```

# BIND

A função bind em vez de executar imediatamente o método que estamos “copiando” para dentro de um objeto, guarda a referência em memória para que seja reutilizado.

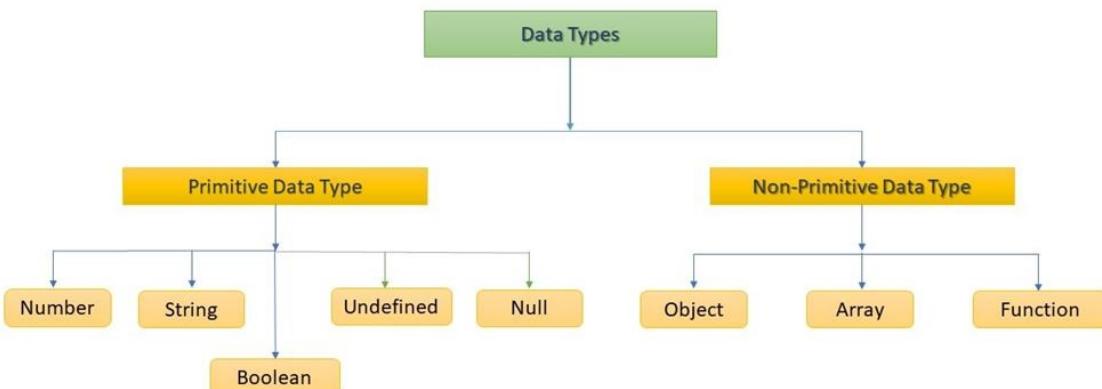
```
● ● ●  
const people = {  
  name: 'Gabriel',  
  energy: 50,  
  setEnergy(n){  
    return this.energy += n;  
  }  
};  
const people2 = {  
  name: 'Gabriel',  
  energy: 50  
}  
const healPeople2 = people.setEnergy.bind(people2, 10);  
healPeople2();  
healPeople2();  
console.log(people2);  
// { name: 'Gabriel', energy: 70 }
```

Uma coisa interessante que podemos fazer, é guardar referências de parâmetros únicas

```
● ● ●  
function multiply(a, b){  
  return a*b  
}  
  
const multiplyByTwo = multiply.bind(this, 2);  
multiplyByTwo(4);  
//8  
const multiplyByFour = multiply.bind(this, 4);  
multiplyByFour(8);  
//32
```

# TYPES

No Javascript temos dois grupos de tipos, os primitivos e os não primitivos,



segue abaixo os devidos tipos e como eles são inicializados.

```
● ● ●  
//Primitivos  
typeof 5  
typeof true  
typeof 'To be or not to be'  
typeof undefined  
typeof null  
typeof Symbol('just me');  
  
//Não primitivos  
typeof {}  
typeof []  
typeof function(){}  
● ● ●
```

é de extrema importância ter que consciência de que arrays e funções também são objetos. Então é por isso que absolutamente todas as estruturas fornecidas pelo Javascript vem de objetos que formam outros objetos (falaremos mais em herança de protótipo).

## PASSAGEM POR VALOR E REFERÊNCIA

Existem duas formas para repassar valores, primeiro trataremos de valores:

```
● ● ●  
var n1 = 1;  
var n2 = n1;  
  
console.log(n2) // => 1;  
● ● ●
```

E é simplesmente isto, quando atribuímos uma cópia de um valor de uma variável para que seja utilizada em outra. Isto é bem simples, vamos com o próximo exemplo usando passagem por referência:

```
● ● ●  
const first = [1, 2];  
const second = first;  
  
second.push(3);  
console.log(first)  
//[1, 2, 3];  
console.log(second);  
//[1, 2, 3];  
● ● ●
```

observe que, ao inserir um novo valor no array ‘second’, o mesmo valor foi inserido no outro array ‘first’, isso porquê ao atribuir um com o outro utilizando a igualdade, ambos guardarão os mesmos valores por estarem compartilhando de uma mesma posição em memória.

O mesmo acontece com objetos

```
● ● ●

const obj1 = {
  n: 10,
}

const obj2 = obj1;
obj2.n = 50000;

console.log(obj1)
// => { n: 50000 };
```

há duas maneiras para ‘clonar’ um objeto de forma ‘exclusiva’.

1.

```
● ● ●

const obj = {
  n: 10
}

const clone = Object.assign({}, obj);
clone.n = 50000;

console.log(obj);
// => { n: 10 }
```

2.

```
● ● ●

const obj = {
  n: 10
}

const clone = {...obj};
clone.n = 50000;

console.log(obj);
// => { n: 10 }
```

# FUNÇÕES SÃO OBJETOS

Antes de nos aprofundarmos mais nos dois pilares do Javascript, precisamos reiterar o conceito de que “tudo” no ambiente é baseado em um objeto, por exemplo as funções:

```
● ● ●  
const say = new Function("console.log('uhuul')");  
  
say.myVar = 'I am a var insided say object';  
  
console.log(say.myVar) //=> I am a var insided say object;
```

```
● ● ●  
function a(){  
  console.log('uhuul');  
};  
  
a.myVar = 'I am a var insed a object';  
  
console.log(a.myVar) // => I am a var insed a object;
```

Ok, agora que te provei que funções são uma espécie de objetos especiais que recebem métodos de manipulação em sua construção como call, apply, bind, podemos entrar em um acordo conceitual: já que podemos repassar objetos dentro de objetos e já que objetos são funções, significa que também podemos passar funções dentro de funções, certo? Certo.

## FIRST CLASS CITIZENS

Na seção anterior, falamos em como as funções podem ser retransmitidas, e isto vem de um conceito que as pessoas chamam de ‘first class citizen’ ou funções são uma primeira classe do Javascript. O que isto significa? Bom, três coisas

1. Funções podem ser declaras em variáveis

```
● ● ●  
var fnc = function(){};
```

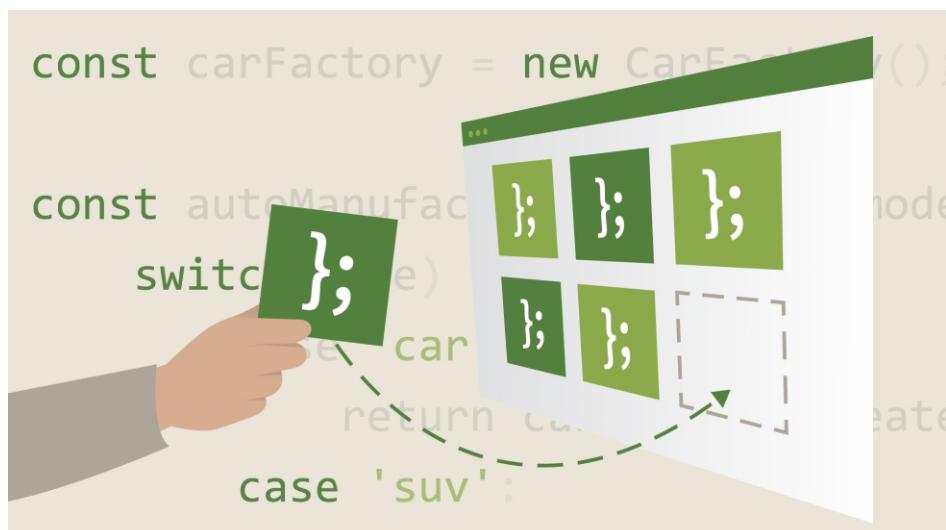
2. Funções podem ser passadas para dentro de outra

```
● ● ●  
var a = function(fn){  
}; fn()  
  
a(function(){ console.log('hi there') }) // => 'hi there';
```

### 3. Podemos retornar funções dentro de outras funções

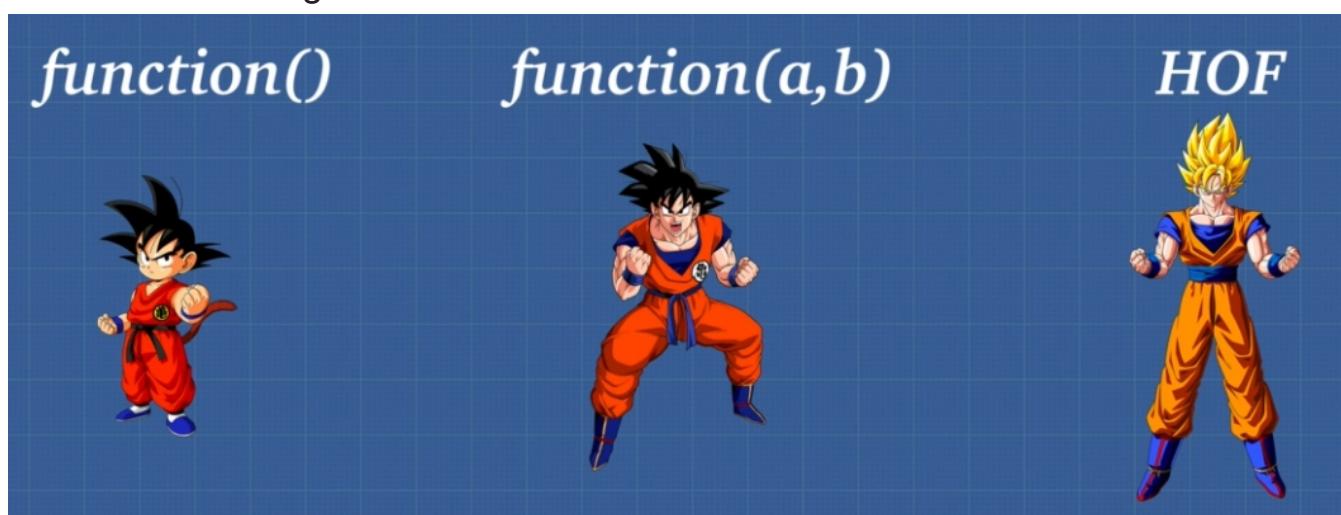
```
var a = function(){  
  return function b(){  
    console.log('I am function b');  
  };  
  
var b = a();  
  
b() // => 'I am function b';
```

Bom, essas três propriedades tornam uma função cidadã de uma primeira classe, ou uma função sendo um dado acoplado a um ‘objeto’ e esses dados não somente executam ações, mas também podem armazenar pedaços de outros dados que podem ser repassados.



## HIGH ORDER FUNCTIONS

Antes de aprofundarmos de fato no que realmente uma HOF é, precisamos diferenciar os diferentes tipos de funções que escrevemos habitualmente. Antes de começarmos observe essa imagem:



*Observação: as definições a seguir não se encontram na literatura, em exceção das ‘HOF’s. Utilizaremos exemplos simples, para que fique mais claro a usabilidade das funções de ordem.*

### 1. ‘funções estaticamente exclusivas’

essas funções, sabem exatamente o que fazer, com dados estáticos. Isto é, sempre farão o mesmo trabalho com os mesmos dados. Para exemplificar, imagine um sistema de login, e neste sistema eu tenho uma função exclusiva que irá efetuar a autenticação no sistema de cada usuário.

```
● ● ●

1 function loginGaba(){
2   let array = [];
3   for(let i = 0; i < 100000000; i++){
4     array.push(i);
5   }
6
7   return 'Access granted to Gaba'
8 }
9
10 loginGaba() // => 'Access granted to Gaba';
```

O laço de repetição é só uma forma para simularmos um trabalho mais pesado que aconteceria em situações reais de login.

Agora imagine que caso quisermos autenticar outros usuários, teríamos que criar novas funções.

```
● ● ●

1 function loginGaba(){
2   let array = [];
3   for(let i = 0; i < 100000000; i++){
4     array.push(i);
5   }
6
7   return 'Access granted to Gaba'
8 }
9
10 loginGaba() // => Access granted to Gaba;;
11
12 function loginMelissa(){
13   let array = [];
14   for(let i = 0; i < 100000000; i++){
15     array.push(i);
16   }
17
18   return 'Access granted to Melissa'
19 }
20
21 loginMelissa() // => Access granted to Melissa;
22
```

## 2. funções dinamicamente exclusivas.

Ok, não é nem um pouco viável toda essa repetição de código, podemos modernizar isso utilizando passagem de parâmetros, logo nossa função saberá exatamente o que fazer independente de quais dados se tratam.

```
● ● ●

1 const giveAcessTo = (name) =>
2   'Access greanted to ' + name
3
4 function login(user){
5   let array = [];
6   for(let i = 0; i < 100000000; i++){
7     array.push(i);
8   }
9
10  return giveAcessTo(user)
11 }
12
13 login('gaba');
```

## 3. Funções de ordem superior (High order functions).

Finalmente chegamos no ponto chave aqui, anteriormente a ordem que davamos as funções era de quais dados utilizar, já com ‘high order functions’ diremos o que a função exatamente deve fazer.

primeiro vamos refatorar o código anterior, fazendo uma leve alteração na sua lógica, impondo um sistema de login por tipo.

```
● ● ●

1 const giveAcessTo = (name) =>
2   'Access greanted to ' + name
3
4 function authenticate(n){
5   let array = [];
6   for(let i = 0; i < n; i++){
7     array.push(i);
8   }
9
10  return true;
11 }
12
13 function login(user){
14   if(user.level === 'admin'){
15     authenticate(5000000);
16   } else if(user.level === 'user'){
17     authenticate(100000);
18   }
19
20  return giveAcessTo(user.name)
21 }
22
23 login({ name: 'Gaba', level: 'admin' });
24
```

como pode observar temos dois tipos de login, usuário e admin, no tipo de admin o laço de repetição é executado por mais tempo só para simular maiores verificações, como aconteceria em situações reais. Agora finalmente vamos implantar a nossa HOF.

```
● ● ●
1 const giveAcessTo = (name) =>
2   'Access greanted to ' + name
3
4 function authenticate(n){
5   let array = [];
6   for(let i = 0; i < n; i++){
7     array.push(i);
8   }
9
10  return true;
11 }
12
13 function login(user, fn){
14   if(user.level === 'admin'){
15     fn(5000000);
16   } else if(user.level === 'user'){
17     fn(100000);
18   }
19
20  return giveAcessTo(user.name)
21 }
22
23 login({ name: 'Gaba', level: 'admin' }, authenticate);
```

Observe que neste momento, passamos a função que deve ser executada como parâmetro, e executamos ela de acordo com a condicional. Isto mostra a eficiência que podemos impor dentro da construção da lógica de nossas funções abstraindo ao máximo cada execução.

## CLOSURES

Finalmente chegamos em um dos pilares do Javascript, ‘fechamento’ mais conhecido como ‘Closures’, ele é baseado em uma aparelhagem envolvendo o Lexical Scope.



*function()*

+



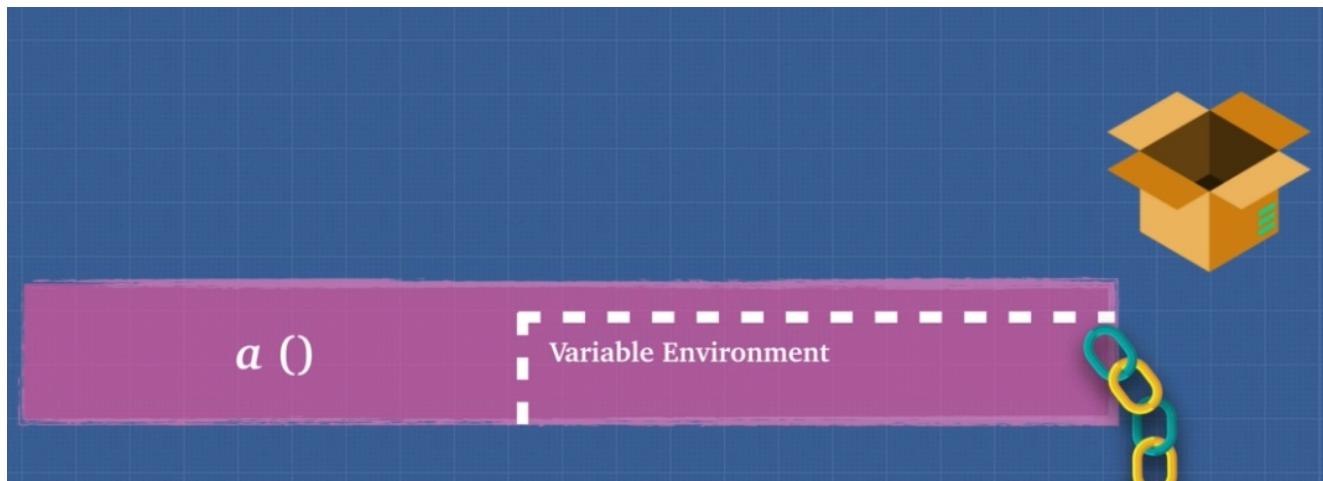
*Lexical scope*

sabemos que antes mesmo do nosso código ser executado, o Javascript lê o nosso ambiente e cria diferentes escopos, cada escopo tem acesso as suas próprias variáveis de acordo com cada ‘mundo’. Observe o código abaixo:

```
1 function a(){
2   let grandpa = 'grandpa';
3   return function b(){
4     let father = 'father';
5     return function c(){
6       let son = 'son';
7       return `${grandpa} ${father} ${son}`
8     }
9   }
10 }
11
12 a()()() // => grandpa father son;
```

Temos uma função que vai retornando outra sucessivamente até chegar na função C, que retorna os dois valores presentes em seus ‘pais’ e o seu próprio valor. Pensando de forma sistemática, função ‘A’ e ‘B’ foram executadas, então como a função C consegue pegar valores de funções que já não existem mais no contexto de execução?

Bom já sabemos que existem link’s para diferentes escopos, baseado nisto temos uma espécie de ‘coletor de lixo’ ou espaço em ‘cache’ que verifica se aquele dado, daquele ambiente está sendo acesso por outro escopo, se sim, este mesmo dado é guardado para que seja buscado, mesmo depois do ambiente ‘principal’ ser quebrado.



```
1 const sayNames = (name1) => (name2) => (name3) =>
2   console.log(` ${name1} ${name2} ${name3}`);
3
4 sayNames('Gabriel')('Cerqueira')('Moraes');
5 // => Gabriel Cerqueira Moraes
```

nós vamos encadeando, linkando uma coisa quando utilizamos um dado em um escopo fora do qual estamos, o Javascript comprehende e isso é guarda exatamente a referência do que está sendo utilizado em memória.

# CLOSURES AND MEMORY

Utilizando a técnica de Closures, conseguimos ter um ganho significado de memoria evitando que um mesmo dado considerado pesado seja recriado toda vez que sua função pertencente for executada. Siga com este exemplo:

```
● ● ●  
1 function heavyDuty(index){  
2   const bigArray = new Array(7000).fill(':D');  
3   console.log('created!');  
4   return bigArray[index];  
5 }  
6  
7 heavyDuty(677);  
8 // => created  
9 heavyDuty(677);  
10 // => created  
11 heavyDuty(677);  
12 // => created  
13 heavyDuty(677);  
14 // => created
```

Observe que toda vez que invocamos a função 'heavyDuty' um novo 'bigArray' é criado gerando uma perda abusrda de performance. Seguindo o conceito que aprendemos podemos resolver esse problema da seguinte forma:

```
● ● ●  
1 function heavyDutyClean(){  
2   const bigArray = new Array(7000).fill(':D');  
3   console.log('created!');  
4   return function(index){  
5     return bigArray[index];  
6   }  
7 }  
8  
9 const script = heavyDutyClean();  
10 // => created  
11  
12 script(677);  
13 script(677);  
14 script(677);  
15 script(677);  
16 script(677);
```

veja que agora guardamos a função responsável por acessar o conteúdo do array, dessa forma estamos ordenando ao Javascript que guarde a referência do conteúdo em memória para que utilizemos o dado toda vez que for necessário sem a necessidade de ter que recria-lo.

# CLOSURES ENCAPSULAMENT

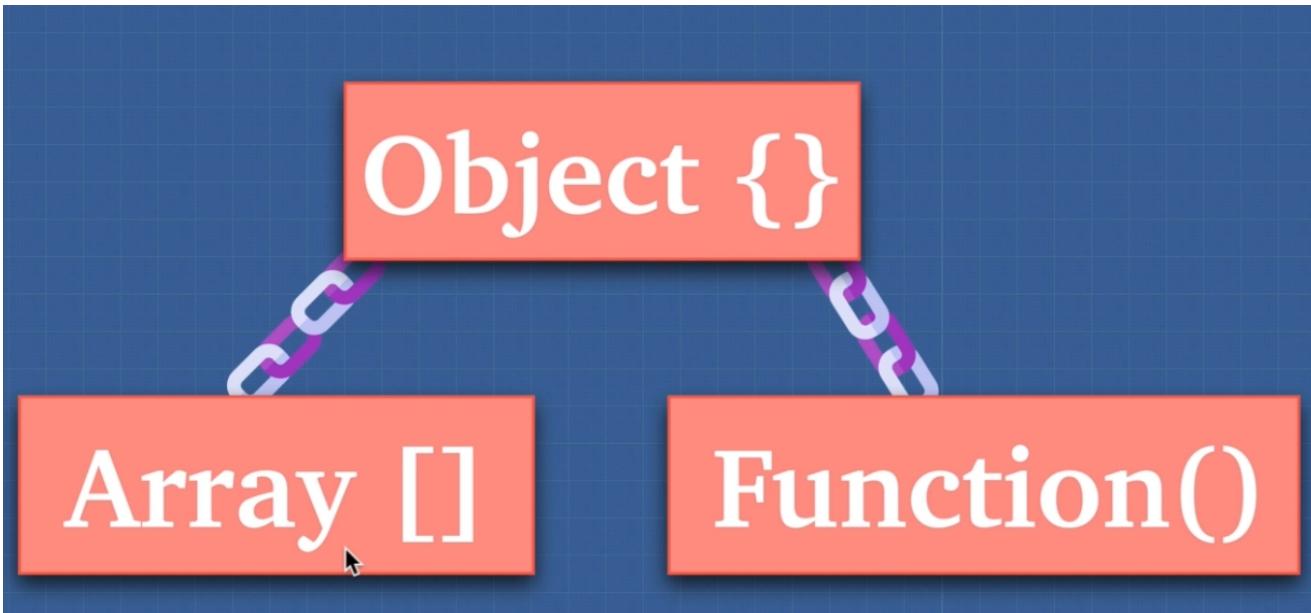
Imagine uma função que simula uma bomba que contém um contador de um segundo, e para cada tick temos uma variável que irá armazenar quanto tempo nossa bomba ficou sem explodir. Agora imagine que queremos escrever essa função de forma encapsulada, fora do contexto global. Com absoluta certeza posso dizer que em meios tradicionais isso se torna dificultoso, mas o closures pode ser nossa solução

```
● ● ●  
1 const makeNuclearButton = () => {  
2   let timeWithoutDestruction = 0;  
3   const passTime = () => timeWithoutDestruction++;  
4   const totalPeaceTime = () => timeWithoutDestruction;  
5  
6   const launch = () => {  
7     timeWithoutDestruction = -1;  
8     return 'boom';  
9   }  
10  
11   setInterval(passTime, 1000);  
12  
13   return {  
14     launch: launch,  
15     totalPeaceTime: totalPeaceTime  
16   }  
17 }  
18  
19 const hono = makeNuclearButton();
```

Logo toda vez que acessarmos a ‘propriedade’ launch que é uma função, o nosso contador irá ser zerado, e vale o mesmo para ‘totalPeaceTime’, toda vez que o buscarmos, será retornado para nós o exato valor de acordo com o tempo decorrido. Explêndido, afinal estamos guardando o retorno propriedades em um manipuladore que por sua vez terá o valor final singular, ‘cacheado’, gerando um retorno dinâmico e não estático.

# PROTOTYPAL ENHANRITENCE

Enfim chegamos no segundo e último pilar do Javascript, este é responsável por criar toda estrutura de herança como conhecemos. Vamos voltar um pouco no tempo e relembrar que tudo no Javascript é baseado em objetos sejam os próprios objetos, funções e arrays. Agora observe a seguinte imagem:



Considere que este objeto no topo, é um objeto especial responsável por construir alguns de nossos elementos, como arrays e funções por exemplo, e toda vez que estes mesmos elementos são criados, eles herdam props presentes neste ‘objeto especial’. Agora vamos criar um elemento e verificar que o responsável por efetuar essa linkagem de pai para filho é chamado de ‘`__proto__`’.

```
> const myArr = [];
< undefined
> myArr.__proto__
< [constructor: f, concat: f, copyWithin: f, fill: f, find: f, ...] 🔍
  ▶ concat: f concat()
  ▶ constructor: f Array()
  ▶ copyWithin: f copyWithin()
  ▶ entries: f entries()
  ▶ every: f every()
  ▶ fill: f fill()
  ▶ filter: f filter()
  ▶ find: f find()
  ▶ findIndex: f findIndex()
  ▶ flat: f flat()
  ▶ flatMap: f flatMap()
  ▶ forEach: f forEach()
  ▶ includes: f includes()
  ▶ indexOf: f indexOf()
  ▶ join: f join()
  ▶ keys: f keys()
  ▶ lastIndexOf: f lastIndexOf()
  length: 0
  ▶ map: f map()
  ▶ pop: f pop()
```

Veja que acessamos um objeto, sendo ele pai do nosso array ‘myArr’, e absolutamente todos os nossos arrays no Javascript herdam deste mesmo objeto que nos fornece funções úteis para manipulação. Lembrando que este objeto não se trata do objeto especial que comentamos anteriormente, mas por ‘sorte’ conseguimos visualizar ele já que o objeto construtor de arrays obrigatoriamente herda as informações dele.

```
> myArr.__proto__.__proto__
< [constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...] 🔍
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()
```



```
1 const fighters = {
2   attack(){
3     return `${this.name} attacked with ${this.force}`
4   },
5
6   defense(){
7     return `${this.name} defended with ${this.protect}`
8   }
9 }
10
11 const ronda = {
12   name: 'Ronda',
13   category: 'medium',
14   force: 75,
15   agility: 95,
16   protect: 53
17 }
18
19 ronda.__proto__ = fighters;
20
21 ronda.attack();
22 // => Ronda attacked with 75
```

Observe que pusemos dentro do objeto que representa o pai do objeto ‘ronda’ usando o ‘`__proto__`’, as funções internas a ‘fighters’. Logo nossa lutadora ronda se procriou dos métodos ‘attack()’ e ‘defense()’.

Vamos agora inspecionar um pouco a herança a partir de nossas funções que criamos rotineiramente.

```
> function myFc(){ };
< undefined
> myFc.__proto__
< f () { [native code] }
>
```

veja que para toda função criada, nos é herdado uma outra função nativa que por sua vez herda as mesmas informações do nosso ‘objeto especial’.

```
> myFc.__proto__.__proto__;
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __LookupGetter__: f, __}
  > constructor: f Object()
  > hasOwnProperty: f hasOwnProperty()
  > isPrototypeOf: f isPrototypeOf()
  > propertyIsEnumerable: f propertyIsEnumerable()
  > toLocaleString: f toLocaleString()
  > toString: f toString()
  > valueOf: f valueOf()
  > __defineGetter__: f __defineGetter__()
  > __defineSetter__: f __defineSetter__()
  > __LookupGetter__: f __LookupGetter__()
  > __LookupSetter__: f __LookupSetter__()
  > get __proto__: f __proto__()
  > set __proto__: f __proto__()
```

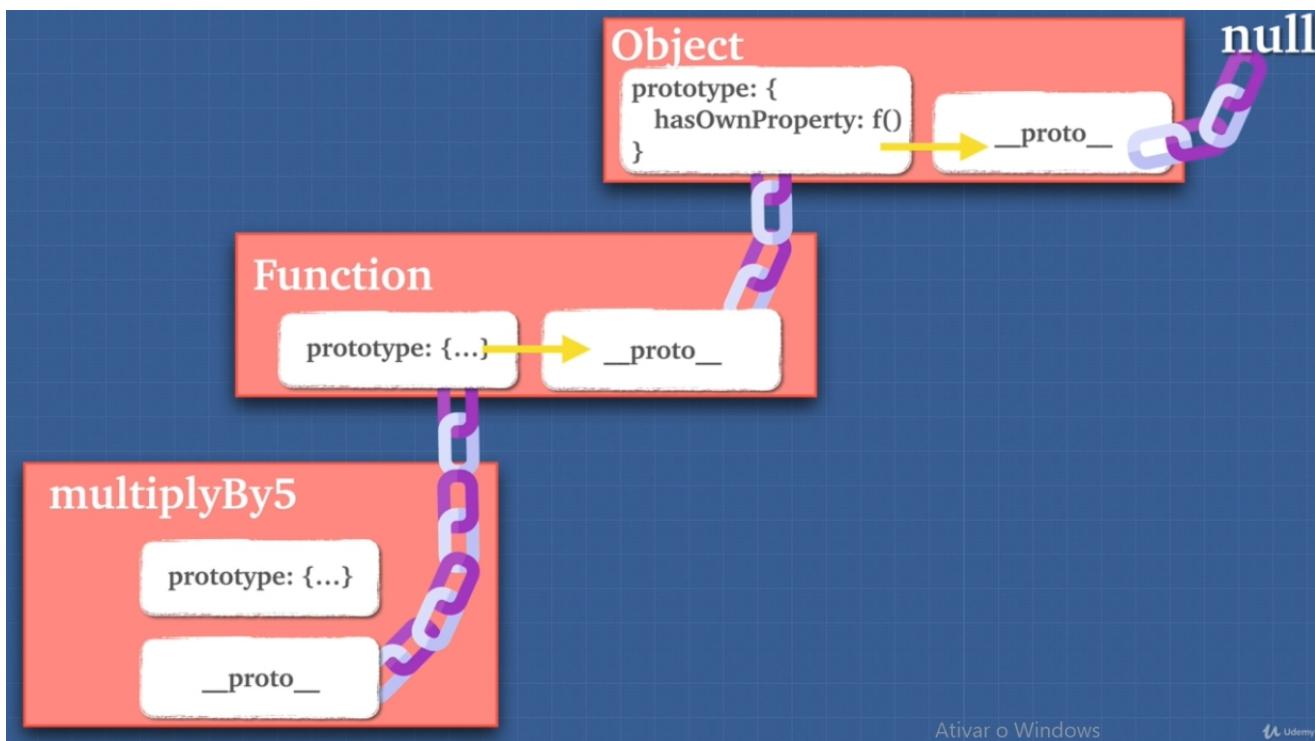
acredito que você tenha assimilado a ideia de como criar uma estrutura de herança usando objetos. Mas quero convencer a você a reconsiderar o meio que utilizamos anteriormente para outro método, isto porquê ele pode causar alguns conflitos durante a criação de múltiplos protótipos de herança e para evitá-los basta utilizarmos a função ‘Object.create’.



```
1 let human = {  
2   mortal: true  
3 }  
4  
5 let socrates = Object.create(human);  
6  
7 socrates.mortal;  
8 //=> true
```

```
> let human = {  
    mortal: true  
}  
  
let socrates = Object.create(human);  
  
socrates.mortal;  
//=> true  
< true  
> socrates.__proto__  
< > {mortal: true} ┌─┐  
  mortal: true  
  ───────────► __proto__: Object  
>
```

Agora que realmente sabemos isto, vamos voltar a inspecionar a herança no Javascript para solidificar nosso conhecimento. Observe esta imagem:



Observe que o primeiro bloco se refere a uma função que criamos ‘multiplyBy5’, e este por ser uma função automaticamente herda de uma outra função nativa imposta pelo próprio Javascript. Essa função nativa por sua vez herda informações do ‘objeto especial’ responsável por construir o pilar de todos os dados que criamos ao longo do desenvolvimento.

olhe também que como já aprendemos, o responsável por criar esta conexão de ‘pai para filho’ é o ‘`__proto__`’, mas veja que há uma outra propriedade que ainda não comentamos chamada ‘prototype’, esta é responsável por fornecer atributos para seus respectivos ‘filhos’. Para exemplificar, vamos acessar o objeto global Date que usamos para manipular datas da seguinte forma:

```
> Date.prototype
< ▷ {constructor: f, toString: f, toDateString: f, toTimeString: f, toISOString: f, ...} □
  ▷ constructor: f Date()
  ▷ getDate: f getDate()
  ▷ getDay: f getDay()
  ▷ getFullYear: f getFullYear()
  ▷ getHours: f getHours()
  ▷ getMilliseconds: f getMilliseconds()
  ▷ getMinutes: f getMinutes()      function getMilliseconds() { [native code] }
  ▷ getMonth: f getMonth()
  ▷ getSeconds: f getSeconds()
  ▷ getTime: f getTime()
  ▷ getTimezoneOffset: f getTimezoneOffset()
  ▷ getUTCDate: f getUTCDate()
  ▷ getUTCDay: f getUTCDay()
  ▷ getUTCFullYear: f getUTCFullYear()
  ▷ getUTCHours: f getUTCHours()
  ▷ getUTCMilliseconds: f getUTCMilliseconds()
  ▷ getUTCMinutes: f getUTCMinutes()
  ▷ getUTCMonth: f getUTCMonth()
  ▷ getUTCSeconds: f getUTCSeconds()
  ▷ getYear: f getYear()
  ▷ setDate: f setDate()
  ▷ setFullYear: f setFullYear()
  ▷ setHours: f setHours()
  ▷ setMilliseconds: f setMilliseconds()
  ▷ setMinutes: f setMinutes()
```

Logo aí está todos métodos que ‘recebemos’ ao criar uma Instância de uma nova Data,

```
> const myDate = new Date('2001-04-25');
< undefined
> myDate.__proto__;
< ▷ {constructor: f, toString: f, toDateString: f, toTimeString: f, toISOString: f, ...}
> myDate.getMilliseconds
< f   constructor      Date
  getDate
  getDay
  getFullYear
  getHours
  getMilliseconds
  getMinutes
  getMonth
  getSeconds
  getTime
  getTimezoneOffset
```

pois além de criar este apontamento para o objeto global Date, nos é fornecido justamente todos essas funções de manipulação para a própria data.

Ok, agora vamos finalizar corrigindo, ou complementando uma informação de que: somente funções carregam em si mesmo a propriedade ‘`__proto__`’. E com base nisto, você me pergunta “mas você não disse que este Date, era um ‘objeto global? Como um objeto carrega essa propriedade?’”, a resposta é relativamente simples, Date é um tipo especial de função, assim como o nosso objeto especial que diretamente ou indiretamente é herdado por todos nossos objetos, funções etc.

```
> typeof Date
< "function"
```

Mas no fim, você também pode considerar ao pé da letra que funções ainda sim são objetos. rsrs

# OOP (ORIENTED OBJECT PROGRAMMING )

Programação orientada a objetos segue um conceito de acoplamento de dados e funções que já aplicamos no tópico anterior quando abstraímos nossos elementos para dentro de objetos e seguimos uma orientação de herança, isto nos permitiu escrever códigos mais limpos sem a necessidade de reescrita. Porém, convenhamos que a utilização constante da forma como aprendemos pode acabar se tornando cansativo e mais complicado a medida em que a complexidade dos protótipos de herança aumentam, então pensando nisso surge as classes no 'Es6' visando facilitar o nosso trabalho ao utilizar OOP. Mas antes de te apresenta-lo precisamos entender como ele funciona 'por de baixo dos panos'.

## CONSTRUCTOR FUNCTIONS

Nossos construtores de funções, são função que ao serem utilizadas junto a nova keyword 'new', retornarão um objeto que será um protótipo de herança desta mesma função. Na prática teremos isto:

```
● ● ●  
1 function Heroes(force, power){  
2   this.force = force;  
3   this.power = power;  
4 }  
5  
6 const superMan = new Heroes(75, 100);  
7  
8 console.log(superMan);  
9 // () => Heroes { force: 75, power: 100 }  
10
```

Veja que agora 'superMan' se tornou um objeto com o auxílio da função construtora 'Heroes', sendo isto possível através da keyWord new.

Repare que a partir de agora, 'superMan' virou um protótipo de herança da 'função', e caso acessemos de dentro de superman a propriedade que nos leva ao pai, teremos a própria função.

```
● ● ●  
1 function Heroes(force, power){  
2   this.force = force;  
3   this.power = power;  
4 }  
5  
6 const superMan = new Heroes(75, 100);  
7  
8 console.log(superMan.__proto__);  
9 // () => Heroes {}  
10
```

relembrando um conceito importante, somente funções carregam em si mesmos a propriedade ‘prototype’, podemos reforçar isto com o mesmo exemplo.

```
> function Heroes(force, power){  
  this.force = force;  
  this.power = power;  
}  
  
const superMan = new Heroes(75, 100);  
  
console.log(Heroes.prototype);  
  
▼ {constructor: f} □  
  ▷ constructor: f Heroes(force, power)  
  ▷ __proto__: Object
```

E sob este mesmo conceito de ‘prototype’ e considerando que este é responsável por distribuir as informações ao seus filhos, incrementaremos nele a seguinte função:



```
1 function Heroes(force, power){  
2   this.force = force;  
3   this.power = power;  
4 }  
5  
6 Heroes.prototype.attack = function(){  
7   return `attack with ${this.power} power`;  
8 }  
9  
10 const superMan = new Heroes(75, 100);  
11  
12 superMan.attack();  
13 // () => attack with 100 power
```

Considerando que a ‘keyword’ this aponta para o objeto na qual a função é uma propriedade e lembrando sobre o escopo dinâmico, o que precisaríamos fazer para a função abaixo não nos retornar um valor indefinido?



```
1 function Heroes(force, power){  
2   this.force = force;  
3   this.power = power;  
4 }  
5  
6 Heroes.prototype.attack = function(){  
7   function attackWithForce(){  
8     return `attack with ${this.force} force`;  
9   }  
10  return attackWithForce();  
11 }  
12  
13 const superMan = new Heroes(75, 100);  
14  
15 console.log(superMan.attack());  
16 // () => attack with undefined force
```

uma alternativa é usar a cópia de função ‘bind’ para o objeto informado.

```
1 function Heroes(force, power){  
2   this.force = force;  
3   this.power = power;  
4 }  
5  
6 Heroes.prototype.attack = function(){  
7   function attackWithForce(){  
8     return `attack with ${this.force} force`;  
9   }  
10  return attackWithForce.bind(this)();  
11 }  
12  
13 const superMan = new Heroes(75, 100);  
14  
15 console.log(superMan.attack());  
16 // () => attack with 100 force  
17
```

Aproveitamos que a função principal attack tem acesso as propriedades por de fato ser um dado no objeto principal, para passar como parâmetro de execução dentro do método ‘bind’

Seguindo, esta é umas das ‘boas práticas’ que temos até então com métodos de herança, porém essas boas práticas acabam se tornando um verdadeiro desafio quando temos heranças multiplas em Javascript então pensando nisto surje no Es6 um recurso que irá facilitar esses processos através de ‘classes’.

# CLASSES

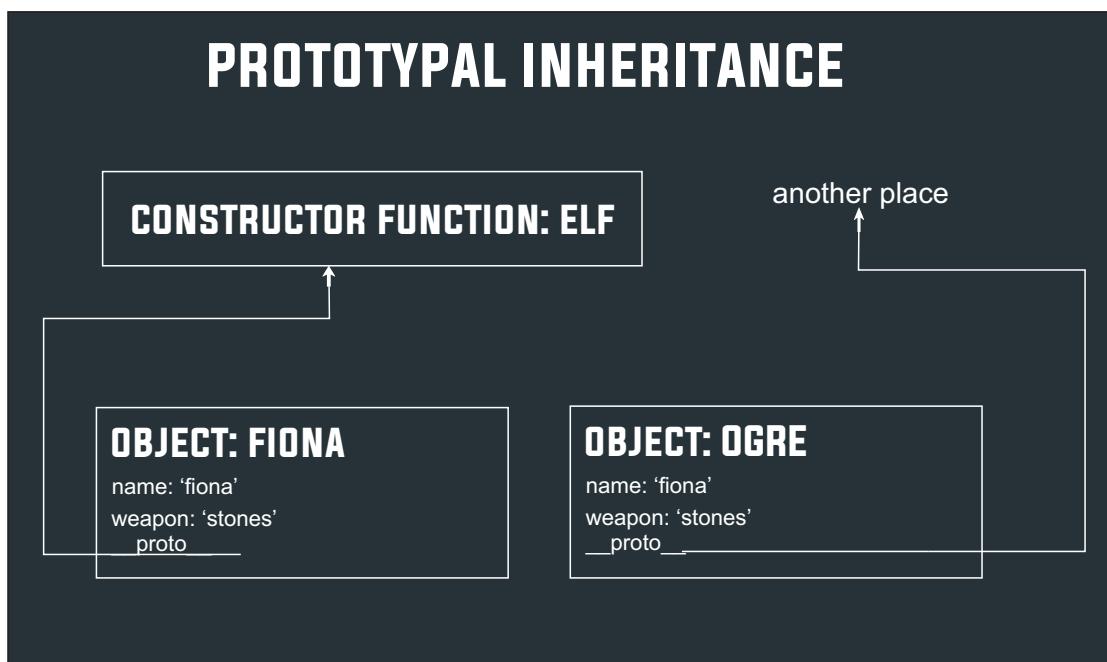
Classes no Javascript surgem com a idéia de simplificar não somente a forma como escrevemos herança mas também um meio para centralizar um código mais limpo. Vejamos agora um exemplo:

```
1 class Elf{  
2   constructor(name, weapon){  
3     this.name = name;  
4     this.weapon = weapon;  
5   }  
6  
7   attack(){  
8     return 'attack with ', this.weapon;  
9   }  
10 }  
11  
12 const fiona = new Elf('fiona', 'stones');  
13  
14 console.log(fiona);  
15 // () => Elf { name: 'fiona', weapon: 'stones' }
```

observe que agora a nossa função construtora surge como uma definição de 'class' e dentro dela temos por padrão um método construtor que pode ser utilizado para receber os parâmetros iniciais e inicializar nossas variáveis/propriedades. Enfim temos um objeto 'fiona' construído de acordo com as informações que nós enviamos, agora vamos copiar todas as propriedades para um outro objeto e verificar as suas paridades.

```
● ○ ●  
1 class Elf{  
2   constructor(name, weapon){  
3     this.name = name;  
4     this.weapon = weapon;  
5   }  
6  
7   attack(){  
8     return 'attack with ', this.weapon;  
9   }  
10 }  
11  
12 const fiona = new Elf('fiona', 'stones');  
13 const ogre = {...fiona}  
14 console.log(fiona === ogre);  
15 // () => false
```

veja, nós copiamos as mesmas informações do objeto 'fiona' para dentro de um novo objeto 'ogre', porém na hora de verificar se ambos são iguais, nós obtemos o valor falso. Isto acontece por um motivo simples, a cadeia de herança dos objetos não são semelhantes. Olhe esta imagem:



Enquanto a nossa propriedade '\_\_proto\_\_' do objeto 'fiona' aponta diretamente para a função construtora (usando um termo mais técnico: 'É um protótipo de herança de Elf') ogro é um simples objeto que não contém as mesmas conexões. Isto explica o valor obtido.

Você talvez esteja se perguntando, 'então se eu instanciar Elf dentro do ogro e verificar se ambos são iguais, obterei um resultado verdadeiro?', a resposta é não.

por mais que ambos passam a ser ‘idênticos’ se tratando de herança e propriedades, o compilador do Javascript os constrói usando propriedades internas diferentes, isto explica o resultado.

Usando nossa mesma base de código, suponhamos que tenhamos outras classes específicas e queremos construir essas classes herdando de uma única classe construtora, já que elas terão propriedades em comum. no fim teremos isto:

```
1 class Character{
2   constructor(name, weapon){
3     this.name = name;
4     this.weapon = weapon;
5   }
6 }
7
8 class Elf extends Character{
9   attack(){
10   return 'attack with ' + this.weapon;
11 }
12 }
13
14 class Ogre extends Character{
15   defense(){
16   return 'defense with ' + this.weapon;
17 }
18 }
19
20 const dolby = new Elf('dolby', 'stones');
21 const shrek = new Ogre('shrek', 'role');
```

veja que agora temos duas instâncias de objetos de classes diferentes, porém ambas as classes herdam atributos parecidos, isto graças á palavra chave ‘extends’ que nos permite herdar dados de uma outra classe, tornando filho dela. Observe que o nosso compilador agora considerou o método construtor da classe pai e isto é o funcionamento padrão. E veja que ao definir dois construtores, obtemos um erro:

```
1 class Character{
2   constructor(name, weapon){
3     this.name = name;
4     this.weapon = weapon;
5   }
6 }
7
8 class Elf extends Character{
9   constructor(){
10 }
11 }
12
13 attack(){
14   return 'attack with ' + this.weapon;
15 }
16 }
17 // () =>
18 //ReferenceError: Must call super constructor in derived class before accessing 'this'
19 or returning from derived constructor
```

O Javascript se perde ao lidar com isto, e em situações como esta, em que queremos utilizar um construtor da classe filha que estamos criando uma instância, basta utilizarmos o método ‘super’, ele indicará que o construtor a ser considerado é o atual e ao mesmo tempo invocará o da classe pai. Observe:

```
● ● ●
1 class Character{
2   constructor(name, weapon){
3     this.name = name;
4     this.weapon = weapon;
5   }
6 }
7
8 class Elf extends Character{
9   constructor(name, weapon, type){
10    super(name, weapon);
11    this.type = type;
12  }
13
14  attack(){
15    return 'attack with ' + this.weapon;
16  }
17 }
18
19
20 class Ogre extends Character{
21   defense(){
22    return 'defense with ' + this.weapon;
23  }
24 }
25
26 const dolby = new Elf('dolby', 'stones', 'stubborn');
27 console.log(dolby);
28 // () => Elf { name: 'dolby', weapon: 'stones', type: 'stubborn' }
29 const shrek = new Ogre('shrek', 'role');
30
```

## POLIMORFISMO

Para finalizar o nosso conhecimento a respeito de classes, vamos tratar sobre polimorfismo, no geral ele é utilizado para descrever um mesmo método que pode ser tratado de diferentes formas usando uma técnica de sobrescrita, vamos com um simples exemplo:

```
● ● ●
1 class Father{
2   speak(){
3     return "none";
4   }
5 }
6
7 class Children extends Father{
8   speak(phrase){
9     return phrase;;
10  }
11 }
12
13 var i = new Children();
14 console.log(i.speak('i am a personal phrase'));
15 // () => i am a personal phrase
```

veja que o método a ser executado foi sobreescrito com uma outra forma funcional. Vamos com outro exemplo:

```
1 class Father{
2     calc(){
3         return 5;
4     }
5 }
6
7 class Children extends Father{
8     calc(){
9         return super.calc() * 5;
10    }
11 }
12
13 var i = new Children();
14 console.log(i.calc());
15 // () => 25
```

veja que desta vez utilizamos a classe que pai presente no ‘Father’ para usar o seu valor de retorno e retornar uma multiplicação, e este é um dos principais pilares do polimorfismo, criar métodos distintos com a capacidade de reúso de funcionalidades que já existem com a possibilidade de ser aprimorada por uma segunda (ou mais) classes.

## FUNCIONAL PROGRAMMING (FP)

Programação funcional não é uma forma de codificar exclusivo á uma linguagem e sim um paradigma com raízes na matemática que trata da separação de interesses que a programação orientada a objeto faz. Tudo se baseia em uma questão de ‘empacotar’ nosso código em pedaços separados e estes pedaços carregam em si a responsabilidade de ser o mais simples possível e realizando o mínimo de tarefas.

## PURE FUNCTIONS

Funções puras é um pilar da programação funcional e cada uma delas devem seguir o conceito de serem simples e exclusivas á uma função como já mencionado, além disso, existem alguns preceitos a serem seguidos vamos comentar um pouco a respeito:

Primeiro de tudo, realizar o mínimo de side-effects o possível.

side-effect é quando usamos a nossa função para se comunicar ou manipular o ‘mundo exterior’, por exemplo:

```
1 const array = [1, 2]
2
3 function pureFunction(){
4     return array.pop();
5 }
6
7 pureFunction();
```

veja que manipulamos um dado fora da função, o que não é considerado uma boa prática, isso porque frequentemente outras funções em paralelo podem estar usando este mesmo estado para manipular algum tipo de informação. O que é muito comum em casos em que múltiplas máquinas estão processando funções, compartilhadas de um mesmo estado, e é exatamente por isso que o estado é imutável rente as funções.

```
1 //good
2 const numbers = [1, 2, 3];
3
4 function immutability(arr){
5   const newArr = arr.concat();
6   newArr.push(':D');
7   return newArr;
8 }
9
10 immutability(numbers);
11 // () => [1, 2, 3, ':D']
12
```

## IDEMPOTENCE

Uma função ‘idempotente’ é aquela que não tem efeito adicional se for chamada mais de uma vez com os mesmos parâmetros de entrada, em outras palavras, a função retornará constantemente os mesmos valores caso seja fornecido os mesmos tipos de parâmetros. Exemplo de uma função não ‘idempotente’:

```
1 //Idempotence
2 function notGood(n){
3   return Math.random(n) * 10;
4 }
5
6 notGood(5);
7 // () => 5.7
8 notGood(5);
9 // () => 10.2
10
```

Exemplo de uma função ‘idempotente’:

```
1 //Idempotence
2 function good(n){
3   return n * 10;
4 }
5
6 good(5);
7 // () => 50
8
9 good(5);
10 // () => 50
```

independente quantas vezes eu chame esta função, ela sempre nos retornará o mesmo valor com base no mesmo parâmetro, sendo assim uma função idempotence.

# IMPERATIVE VS DECLARATIVE

Agora vamos tomar conhecimento de ocorrências imperativas e declarativas, na declarativa as coisas ocorrem de formas menos literárias, por exemplo, na programação ao criarmos uma variável estamos criando uma posição em memória para armazenar aquele valor, mas não estamos dizendo ao compilador em qual posição e como ele deverá ser alocado, logo estamos usando uma forma declarativa para que seja feito. Já na forma imperativa o contrário ocorre, claro que não fazemos isso programando no dia dia, de dizer como as coisas devem ser feitas, pois isto é trabalho do nosso compilador. Para um melhor entendimento, olhe o exemplo abaixo:

```
● ● ●  
1 for(let i = 0; i < 5; i++){  
2   console.log(i)  
3 }  
4  
5 [10, 20, 30].forEach((i) => {  
6   console.log(i);  
7 });
```

o primeiro laço podemos dizer que foi escrito de forma imperativa dado que criamos um laço de repetição especificando detalhadamente o que ele deve fazer e como deve fazer. Já o segundo laço criamos de forma declarativa, pois usamos um recurso do próprio Javascript para percorre-lo de forma mais automatizada.

# CURRYING AND PARTIAL APPLICATION

Currying é o processo de transformar uma função que espera vários argumentos em uma função que espera um único argumento e retorna outra função curried. Por exemplo,

```
● ● ●  
1 const multiply = (a) => (b) => (c) => a*b*c;  
2 const multiplyBy1 = multiply(1);  
3 const multiplyBy2 = multiplyBy1(2);  
4 const multiplyBy3 = multiplyBy2;  
5  
6 multiplyBy3(3);  
7 // () => 6
```

veja que acabamos de particionar os parâmetros, a primeira função recebe um parâmetro que por sua vez retorna uma outra função. Utilizamos a primeira função para guardar a referência da segunda e assim acontece sucessivamente.

agora vamos falar um pouco de partial application, ele é um tanto quanto semelhante ao currying, sendo sua principal diferença a quantidade de argumento que podem ser inseridos antes de uma função ser retornada, algo como isto:

```
1 const multiply = (a, b, c) => a*b*c;
2 const partialMultiply = multiply.bind(null, 5);
3 partialMultiply(8, 10);
4 // () => 400
```

veja que agora nossa função espera receber múltiplos argumentos, e a referência desta vez foi guardada utilizando o método ‘bind’, lembrando que o método ‘bind’ executa uma função de acordo com o objeto informado, neste caso simplesmente passamos o valor nulo por não existir ‘terceiros’.

## MCI MEMOIZATION

Entrando em mais um conceito aplicado na programação funcional, agora temos algo que também podemos chamar de cache. Cache é uma forma de memorizar alguma informação que foi processada, para que as buscas seguintes não precisem de um novo processamento, já que o seu valor já está armazenado e disponível para ser acessado. Vamos com um exemplo:

```
1 function multiplyBy100(number){
2   console.log("Long time");
3   return number * 100;
4 }
5
6 multiplyBy100(100000);
7 // () =>
8 //Long time
9 //10000000
```

criamos uma função, onde sua unica responsabilidade é simplesmente multiplicar um número informado por 100. Considerando que isto é um processamento ‘muito pesado’ e leva certo tempo até que seja feito, seria interessante se nós conseguimos guardar o resultado da multiplicação de um determinado número, para que na próxima execução, não seja necessário calcular este mesmo número novamente e sim buscar na nossa memória cache. Usando closures faremos isto:



```
1 function memoizedMultiplyBy100(){
2   let cache = {
3   }
4 
5   return function(number){
6     if(number in cache){
7       console.log('Absolutely fast', cache[number]);
8       return cache[number];
9     } else {
10      cache[number] = number * 100;
11      console.log('Takes a long time', cache[number]);
12      return cache[number];
13    }
14  }
15 }
16 }
17
18 const memoized = memoizedMultiplyBy100();
19 memoized(10);
20 // () => Takes a long time 1000
21 memoized(10);
22 // () => Absolutely fast 1000
23 memoized(10);
24 // () => Absolutely fast 1000
25
```

repare que na primeira execução, nós verificamos se existe um valor no objeto conforme o número passado, como não existe, caímos no else, realizamos o cálculo e colocando o valor no objeto ‘cache’ sendo o nome da propriedade de acordo com o próprio número, exemplo: { 10: 1000 }.

Já as funções seguintes foram executadas de forma mais ‘rápida’, já que o cálculo de multiplicação o número 10, já tinha sido realizado e guardado em cache, então a nossa função simplesmente recuperou este mesmo valor e imediatamente o retornou sem a necessidade de um novo processamento. AMAZING!

## COMPOSE

Compose é uma forma de compor multiplas funções em uma cadeia de execução, sendo que o resultado de uma é passada como argumento para a próxima. Vamos ver como isto fica:



```
1 const compose = (f1, f2) => (number) => f1(f2(number));
2
3 const multiplyBy2 = (number) => number * 2;
4 const divisorBy2 = (number) => number / 2;
5
6 const calculate = compose(divisorBy2, multiplyBy2);
7 calculate(4);
8 // () => 4
```

ao usar o compose, sempre teremos uma função que irá construir a estrutura de execução das próprias funções. Foi o que fizemos na primeira linha, definimos uma composição, sendo que ela recebe a princípio duas funções como parâmetro e retorna uma outra função que é a execução de ambas. E assim podemos criar a tal cadeia sucessivamente.

```
● ● ●  
1 const compose = (f1, f2, f3) => (number) => f1(f2(f3(number)));  
2  
3 const multiplyBy2 = (number) => number * 2;  
4 const divisorBy2 = (number) => number / 2;  
5 const subtractBy1 = (number) => number - 1;  
6  
7 const calculate = compose(multiplyBy2, divisorBy2, subtractBy1);  
8 calculate(4);  
9 // () => 3
```

## PIPE

O pipe é muito parecido com o compose, sendo sua única diferença a ordem das funções que é dada de forma contrária. Como isto:

```
● ● ●  
1 const compose = (f1, f2, f3) => (number) => f3(f2(f1(number)));  
2  
3 const multiplyBy2 = (number) => number * 2;  
4 const divisorBy2 = (number) => number / 2;  
5 const subtractBy1 = (number) => number - 1;  
6  
7 const calculate = compose(multiplyBy2, divisorBy2, subtractBy1);  
8 calculate(4);  
9 // () => 3
```

por fim vimos alguns pilares básicos na programação funcional onde os principais conceitos estabelecidos até aqui são: a ideia de separação de funções que são puras realizando o mínimo de tarefas o possível seguindo uma imutabilidade de estados, conceito onde o ‘input => output’ é sempre presível, funções retornando valores para uso e manipulação para outras funções seguindo uma ‘cadeia’ lógica sem a necessidade de compartilhamento simultâneo de dados.

Agora com base em nosso aprendizado, vamos criar uma mini aplicação que simulará um carrinho de compras.

Primeiro criaremos uma ‘função construtora’ que será responsável por automatizar aquele processo manual que fizemos recentemente usando o ‘compose’.

```
● ● ●  
1 const compose = (f, g) => (...args) => f(g(...args));  
2  
3 const purchaseItem = (...fns) => fns.reduce(compose);
```

a função ‘purchaseItem’ invocará o método reduce, este irá executar o call back ‘compose’ que seguindo a lógica invocará a função do índice atual e retornará o valor de retorno da função para próxima função (que introduziremos em breve).

```
● ● ●
1 let user = {
2   name: 'Gaba',
3   cart: [],
4   purchases: []
5 }
6
7 const compose = (f, g) => (...args) => f(g(...args));
8
9 const purchaseItem = (...fns) => fns.reduce(compose);
10
11 const addItemToCart = (user, item) => {
12   const updateCart = user.cart.concat(item);
13   return Object.assign({}, user, { cart: updateCart });
14 };
15
16 purchaseItem(
17   addItemToCart
18 )(user, { name: 'ball', price: 200 });
19 // () => {
20 //   name: 'Gaba',
21 //   cart: [ { name: 'ball', price: 200 } ],
22 //   purchases: []
23 // }
```

finalmente criamos o nosso objeto ‘user’, que podemos chamar de estado, e ao invocar a função ‘purchaseItem’ passamos unitariamente todas as funções que necessitamos para manipular a nossa lógica de compras, e como já dito, informamos um parâmetro na ‘função de estrutura’ que nos é retornada na linha 18, e cada função pegará esse parâmetro, manipulará este ‘estado imutável’ e retornará como resultado para próxima função manipular, e assim sucessivamente. E como você também pode observar, sempre criamos um novo objeto evitando a passagem por referência assim como mostra na função ‘addItemToCart’. Para solidificar nosso aprendizado, vamos criar mais algumas funções, primeiro uma para aplicar uma taxa em todos itens do carrinho.



```
1 let user = {
2   name: 'Gaba',
3   cart: [],
4   purchases: []
5 }
6
7 const compose = (f, g) => (...args) => f(g(...args));
8
9 const purchaseItem = (...fns) => fns.reduce(compose);
10
11 const addItemToCart = (user, item) => {
12   const updateCart = user.cart.concat(item);
13   return Object.assign({}, user, { cart: updateCart });
14 };
15
16 const applyTax = (user) => {
17   return Object.assign({}, user, { cart: user.cart.map(product => {
18     product.price *= 1.2
19     return product;
20   }) });
21 }
22
23 purchaseItem(
24   applyTax,
25   addItemToCart
26 )(user, { name: 'ball', price: 200 });
27 // () => {
28 //   name: 'Gaba',
29 //   cart: [ { name: 'ball', price: 240 } ],
30 //   purchases: []
31 // }
```

primeiro criamos a função `applyTax` retornando assinatura de um novo objeto manipulando as informações presentes no ‘cart’, neste caso aplicamos um calculo de 10.2% para cada elemento. Em seguida passamos a função como um dos ‘call backs’ da nossa ‘função de estrutura’.

Agora vamos criar duas funções, uma para esvaziar o nosso carrinho ‘cart’ e passar todos estes elementos para o array de compras ‘purchases’



```
1
2 const buyItem = (user) => {
3   return Object.assign({}, user, { purchases: user.cart });
4 }
5
6 const emptyCart = (user) => {
7   return Object.assign({}, user, { cart: [] });
8 }
9
10 purchaseItem(
11   emptyCart,
12   buyItem,
13   applyTax,
14   addItemToCart
15 )(user, { name: 'ball', price: 200 });
16 // {
17 //   name: 'Gaba',
18 //   cart: [],
19 //   purchases: [ { name: 'ball', price: 240 } ]
20 // }
```

*Predictable*

*1 Task*

*return Statement*

*Composable*



*Pure*

*Immutable State*

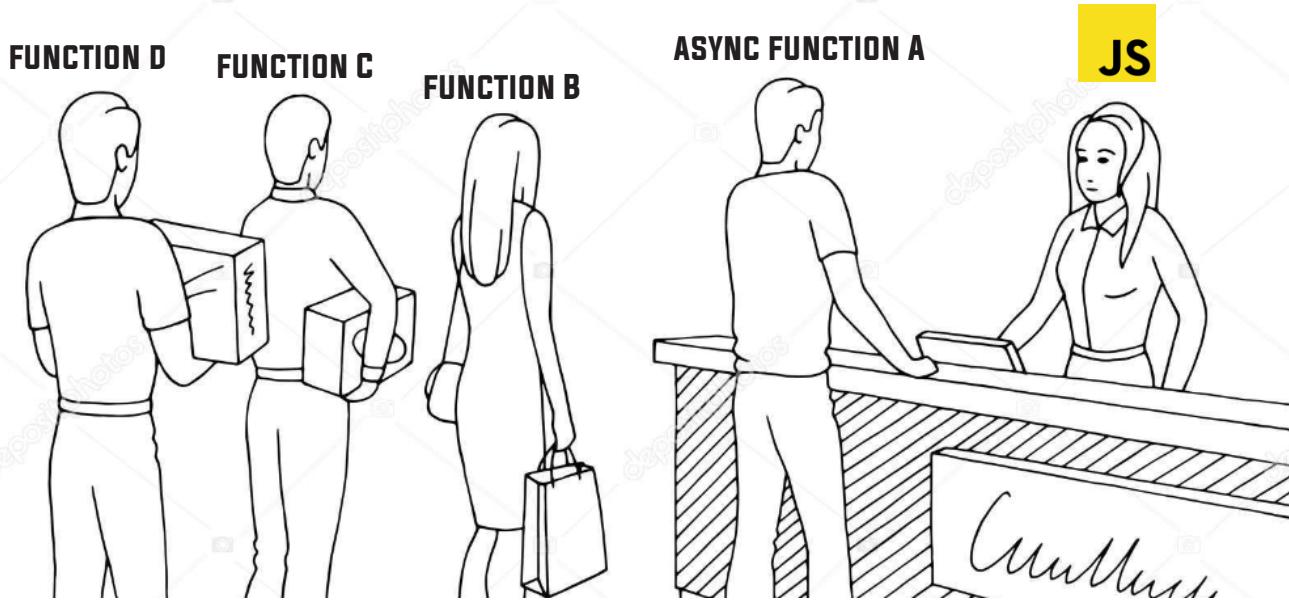
*No shared state*

Udemy

## ASSÍNCRONISMO

Síncrono ou assíncrono diz respeito ao fluxo de execução de um programa. Quando uma operação executa completamente antes de passar o controle à seguinte, a execução é síncrona. Quando uma operação não pode ser concluída de imediato dentro de um contexto de execução, temos uma função assíncrona.

Para exemplificar como de fato são as funções assíncronas, vamos fazer uma analogia com uma fila de pessoas de uma loja, sendo para cada pessoa uma função a ser executada.



Veja que a primeira pessoa da fila (função a ser executada) é uma ‘pessoa’ assíncrona, imagine que essa pessoa ao chegar a sua vez não pode imediatamente pagar pelo ‘produto’ pois algo ocorreu de errado com seu meio de pagamento e para não atrapalhar o andamento da fila, ele ficou ao lado aguardando até que pudesse de fato pagar pelo produto. E é isto o que acontece em um contexto real, funções

assíncronas são executadas em forma de Promises até que de fato estejam prontas. Antes de fato entender como funcionam as Promises, vamos declarar a seguir uma função assíncrona.

```
● ● ●  
1 async function myFirstAsyncFunction() {  
2   try {  
3     const fulfilledValue = await callhttp();  
4     // bloqueado até 'callhttp' retornar uma promise.  
5  
6     console.log(fulfilledValue.data);  
7   }  
8   catch (rejectedValue) {  
9     // ...  
10  }  
11 }
```

Claramente a função de fato não funciona pois misturamos um pouco de pseudocódigo, mas a intenção é somente simular o funcionamento.

Veja que utilizamos a keyword 'await', esta força a espera pelo retorno da função que está sendo invocada á seguir. Uma das 'mágicas' disto é que o contexto de execução interno a função é bloqueado, o que não acontece dentro do contexto em que a função está inserido.

## PROMISES

Uma Promise é um objeto que representa a eventual conclusão ou falha de uma operação assincrona. Essencialmente, uma promise é um objeto retornado para o qual você adiciona callbacks, em vez de passar callbacks para uma função.

Por exemplo, em vez de uma função old-style que espera dois callbacks, e chama um deles em uma eventual conclusão ou falha:

```
● ● ●  
1  
2 function doSomeThing(success, failed){  
3   if(error){  
4     return failed('Ocorreu um erro');  
5   } else {  
6     return success('Sucesso');  
7   }  
8 }  
9  
10 function successCallback(msg){  
11   console.log(msg);  
12 }  
13  
14 function failedCallBack(msg){  
15   console.log(msg);  
16 }  
17  
18 doSomeThing(successCallback, failedCallBack);  
19 // () => Ocorreu um erro  
20
```

com as Promises, podemos simplificar desta forma:

```
1 function doSomething(error){  
2   return new Promise((resolve, reject) => {  
3     if(error){  
4       reject('Ocorreu um erro!');  
5     } else {  
6       resolve('Sucesso!');  
7     }  
8   });  
9 }  
10  
11 doSomething(false).then((msg) => {  
12   console.log(msg);  
13 }).catch((msg) => {  
14   console.log(msg);  
15 });  
16 //Sucesso!
```

A função `doSomething` nos retornou a execução de uma Promise, sendo possível através do nosso construtor de funções. Por sua vez essa ‘Promise’ recebe um callback enviando dois argumentos que são os métodos ‘`resolve`’ e ‘`reject`’, sendo `resolve` representado pelo sucesso da função e `reject` uma falha. Com isto, no retorno de execução de ‘`doSomething`’ podemos lidar com todas as exceções de forma manipulável de acordo com o que de fato aconteceu.