**JAVASCRIPT**

Um Exemplo de como podemos usar aspas no console.log, podemos utilizar aspas, aspas duplas e crases.  
Ex:

console.log(“Paulo ‘Henrique’ ”);  
console.log(‘Paulo “Henrique” ‘);  
console.log(` ’Paulo’ “Henrique” `);

Se utilizarmos crase para exprimir algo no console log, podemos utilizar o template string, que é uma forma de exprimir variáveis sem precisar concatenar com algum determinado texto.  
Ex:  
console.log(`${nome} ${sobrenome} tem ${idade} anos, pesa ${peso} kg`)

**Variáveis**

Em JS é usado let, var ou const para “Variáveis”.

Let é uma forma de var, só que mais nova, mas ainda tem algumas diferenças de escopo, mas no geral é isso.

* Não podemos criar variáveis com palavras reservadas.
* Variáveis precisam ter nomes significativos.
* Não pode começar o nome de uma variável com um número.
* Não podem conter espaços ou traços.
* Utilizamos camelCase.
* Case-sensitive.
* Não podemos redeclarar variáveis com let.
* NÃO UTILIZE VAR, UTILIZE LET.

Constantes

* Não podemos modificar o valor de uma constante.
* Precisa ser criada e inicializada.

Com o var podemos redeclarar a variável diversas vezes, mas com o let, só podemos redeclarar o valor.

Temos vários tipos primitivos, sendo eles: String, Number, Undefined, Null, Boolean e Symbol.

**Undefined**Se inicializamos uma variável sem um valor, ela será undefined, pois ela não irá apontar para nenhum local na memória. O undefined é declarado pelo próprio Js.

**Null**  
Se inicializarmos uma variável com null, ela também será apontada para nenhum local na memória, mas o null é declarado pelo próprio dev.

Uma curiosidade é que o valor null é mostrado como um objeto, mas na realidade é um bug do JavaScript.

Se fizermos uma cópia de uma variável para outra e alteramos o valor da segunda, será alterado o dá primeira, mas se alteramos o valor da primeira, será alterado só o da primeira, pois a segunda é uma cópia da primeira.

O parseInt() Vai fazer com que tal string seja transformada em Number.

O parseFloat() Faz com que uma string seja transformada em Float.

O Number() Não é feito a distinção de valor, então será recolhido qualquer tipo de Número.

**Strings**Podemos manipular as strings de diferentes maneiras, sendo algumas delas mostrada abaixo.

charAt – Retorna o texto de acordo com a posição escolhida na função.  
  
concat – É uma função que faz a concatenação de várias strings.  
  
indexOf – É uma função que procurar por uma determinada palavra dentro de uma variável. E se for usado algum número após a palavra digitada, ele irá buscara do determinado índice indicado.  
  
lastIndexOf - Começa do final para o início da string e é a mesma coisa do indexOf.  
match – É uma função que cria uma array e que pode ser utilizado expressões regulares.  
  
replace – É uma função que faz a troca de letras ou palavras de uma string por outra. É possível utilizar expressões regulares também.  
  
length – Mostra quantos índices tem em uma determinada string.  
  
slice – É uma função que faz o fatiamento de uma string com início e fim.  
  
split – É uma função que faz um recorte dentro de uma string, se for colocado um número logo após, é possível escolher quantas vezes irá fazer esse recorte. Basicamente irá pegar uma string e transformar em um array, e o que for colocado dentro do parâmetro da função, será por onde irá buscar o separador. Se por exemplo tiver um (‘ ‘), será separado por todas palavras que tem um espaço em branco.

**Numbers**Podemos manipular números com diferentes funções.

toString – Transforma um number em string. Se colocarmos um number e o número 2 dentro da função, é possível vermos o número binário do tal.  
  
toFixed – Faz o arredondamento de um número. Dentro da função é preciso escolher quantas casas decimais irá ter depois da virgula.  
  
isInterger – Podemos saber se tal valor é inteiro ou não.  
  
isNaN – Podemos saber se o valor é ou não é um número.

**Math**Math é uma biblioteca nativa do Js que pode ser usada diversas funções.

Math.floor – Arredonda o número para baixo.  
  
Math.ceil – Arredonda o número para cima.  
  
Math.round – Arredonda para o número mais próximo.  
  
Math.max – Pega o maior número de uma sequência.  
  
Math.min – Pega o menor número de uma sequência.  
  
Math.random – Gera um número aleatório.  
  
Math.pow – Faz potenciação.

**Array**É basicamente uma lista, ou uma coleção de coisas.

.shift() - Remove o primeiro elemento do array.

.pop() - Remove o último elemento do array.

.unshift() - Move o elemento para o começo do array.

.push() - Adicionar mais um elemento no final do array.

.slice() – Faz o fatiamento de um array, podendo receber dois valores, sendo o primeiro o inicio do fatiamento e o segundo o final, lembrando que o índice final não é incluído. Podemos utilizar valores negativos.

.join() – Pega um array e transforma em uma string. Dependendo do que for colocado entro do parâmetro, será a forma de separar as palavras.

alunos[alunos.length] = 'Luiza' - Adicionar mais um elemento no final do array.

alunos[0].length) - Tamanho de um elemento no array.

delete alunos[] - Deleta o elemento, mas no lugar dele irá ficar vazio.

Se o array estiver atribuído a uma contante, não será possível alterar a sua forma, um exemplo é: se tiver um array[] como objeto e formos transformar em uma variável normalmente, não é possível mudar, mas se o array estiver em let, é possível.

**Objetos**Os objetos são utilizados igual as outras linguagens. Utilizamos Classes, métodos e atributos, mas por enquanto iremos descobrir um pouco de como utilizar os objetos.



Aqui temos um exemplo de objeto.

Foi criado um objeto pessoa1 com atributos que são basicamente variáveis e podem ser utilizados dentro da classe, sendo eles nome, sobrenome e idade. E também foi criado os métodos, que são basicamente funções e que podem ser executadas várias linhas de códigos.

**Valores primitivos e por referência**

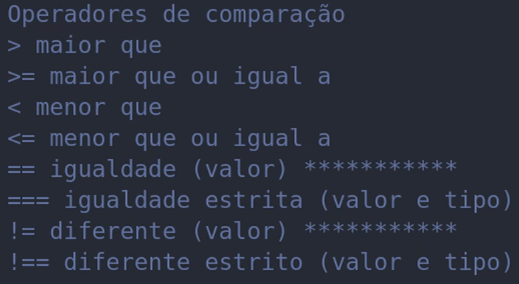
Primitivos (Imutáveis) – string, number, boolean, undefined, null (bigint, Symbol) – Valores Copiados.

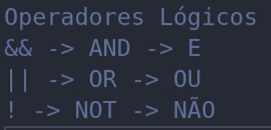
Referência (mutável) – array, object, function – Passados por referência.

Os valores primitivos são imutáveis, ou seja, se for feita a referência de um valor e altera-lo depois, ele apenas será alterado o primeiro valor e não o que foi copiado.

Já os valores por referência são alterados, ou seja, se for feita uma referência do valor e logo após foi alterado o primeiro valor, o segundo será alterado. Caso seja alterado o segundo que foi referenciado do primeiro, também alterara o primeiro valor.

Se colocarmos uma array referenciado de outro e passarmos três pontos e a variável, esse valor não será mexido, pois é uma cópia e não uma referência.

**Operadores de Comparação  
**

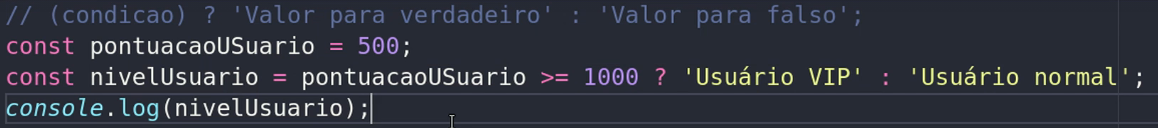
**Operadores Lógicos**

**Curto Circuito (Short-Circuit)**Quando temos uma expressão com vários tipos de verificação, se alguma vez ela for falsa, automaticamente será encerrada, sendo assim gastando menos tempo de verificação.

Se todas as expressões forem verdadeiras, apenas a última será retornada, mas caso tenha alguma falsa no meio, será retornada a última falsa.

Falsy Values – false, 0, aspas simples, duplas e crases, null / undefined e NaN.

**Operação Ternária**É basicamente um if e else compactado.  
Ex:



**DATETIME**Para iniciar a biblioteca date, é preciso criar um construtor em uma constante.  
Ex:  
const data = new Date();

A data é exprimida em Ano / Mês / Dia Hora : Minuto : Segundo : Milissegundo

toString() – Usamos para exprimir a data e a hora atual.

getDate() – Usado para pegar o dia.

getMonth() – Usado para pegar o mês. O Mês começa com 0 e vai até o 11

getFullYear() – Usado para pegar o ano.

getHours() – Usado para pegar as horas.

getMinutes() – Usado para pegar os minutos.

getSeconds() – Usado para pegar os segundos.

getMilliseconds() – Usado para pegar os milissegundos.

getDay() – Usado para pegar o dia da semana, sendo eles 0 – Domingo e 6 – Sábado.

Date.now() – Pega o Dia, Mês, Ano, Horas, Minutos, Segundo e Milissegundos em Milissegundos.

**If, else if e else**Estrutura:

If(condição) {  
código  
}else if (condição) {  
código  
}else {  
código  
}

**Switch case**Estrutura:  
Switch(condição) {  
case 0:  
 código  
 break ou return;  
default:  
 código  
 break ou return;

Podem existir vários casos.

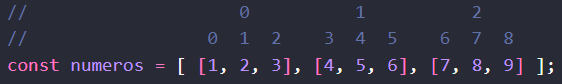
Let tem escopo de bloco {... bloco}.  
Var só tem escopo de função.

**Atribuição e Desestruturação de arrays**A atribuição de arrays é basicamente pegar um array e colocá-los em uma ou várias variáveis de acordo como for selecionado.  
Ex:  


Desse jeito será criado 3 variáveis para cada um dos primeiros elementos e logo após será criada uma variável com o resto dos elementos que sobraram.



Podemos também pular alguns elementos. Se pularmos eles literalmente, eles não serão exibidos.

Agora falando um pouco da desestruturação, é possível pegar arrays dentro de outros de diversas formas.  
Ex:  


Temos esse array e acima temos as posições em que estão, como se fossem dimensões.



Podemos pegar um elemento dessa forma, como se fosse pulando as partes do array com virgulas, mas também tem outra forma.



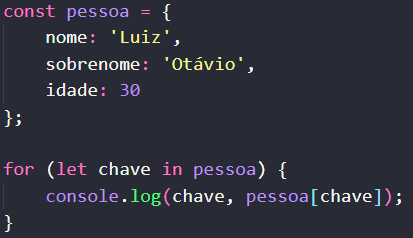
Podemos criar um array para cada elemento maior e logo após selecionar o elemento desejado dentro da primeira dimensão.

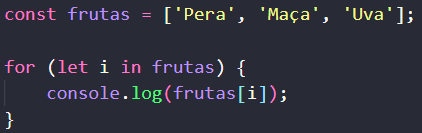


Também podemos fazer desta forma, assim não precisaríamos criar outra variável.

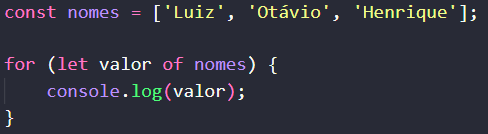
**For**Estrutura:  
For(let i = 0; condição; i++) {  
código;  
}

**For in -**O for in é basicamente um for, mas simplificado. O diferencial é que pode utilizar um objeto.  
Estrutura:  
For (let i in objeto, array ou string) {  
código;  
}

Ex:  




**For of -**O for of é usado para inteiráveis, arrays e string. Será retornado apenas o valor, pois não é capturado o índice.  
Estrutura:  
For (let i of array ou string) {  
código;  
}

Ex:  


**Diferença dos For**

For clássico - Geralmente com iteráveis (array ou strings)  
For in - Retorna o indice ou chave (string, array ou objeto)  
For of - Retorna o valor em si (iteráveis, arrays ou strings)

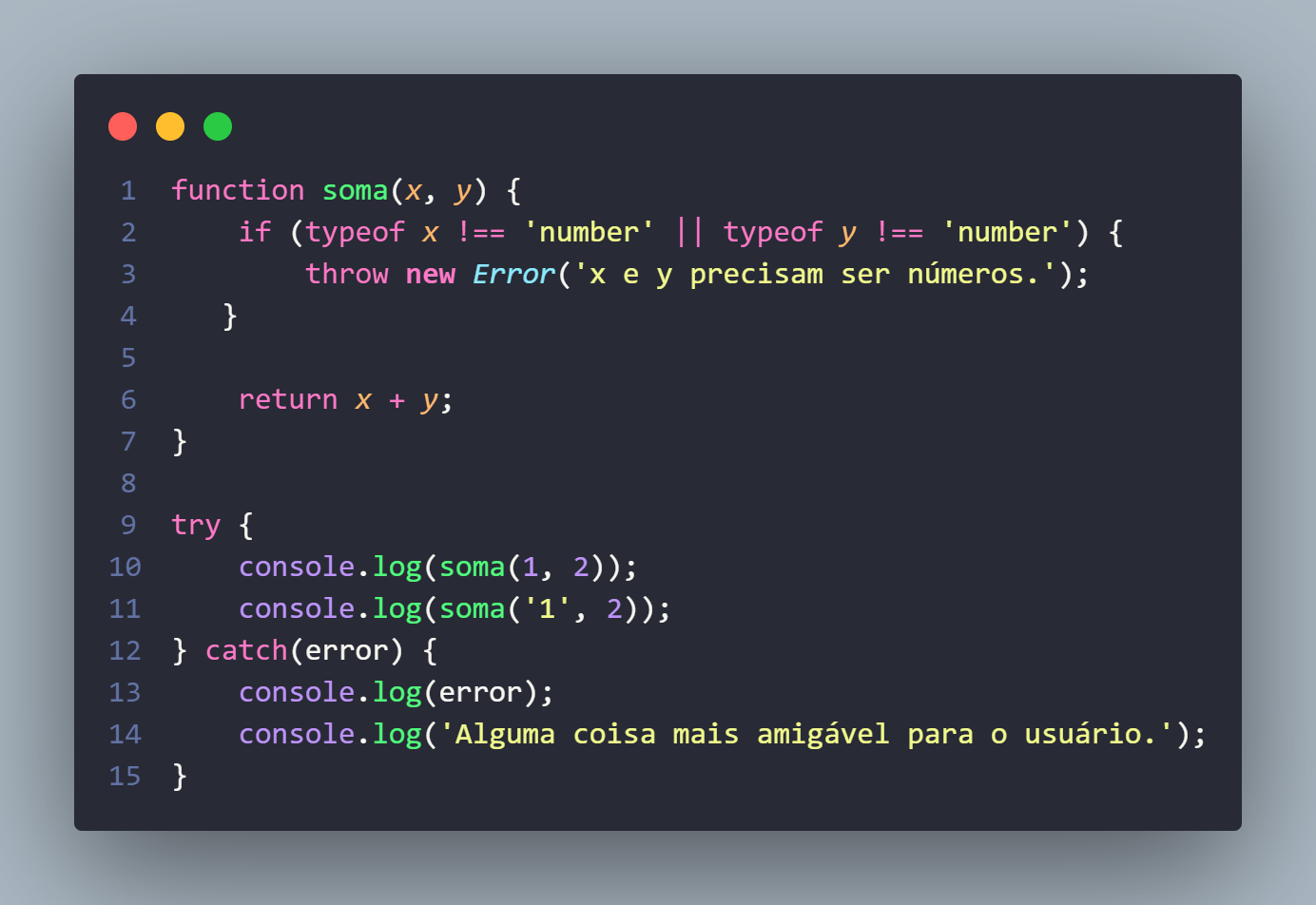
**While**O while é uma estrutura de repetição que, enquanto a condição for verdadeira, o bloco de código será executado. Lembrando que é preciso ter uma variável de contagem, caso contrário, terá um loop infinito. Estrutura:  
while (condição) {  
código;  
}

**Do While -**O do while primeiro executa o código e depois checa a condição, caso seja falsa, será finalizado a execução do código.

**Continue –** Continua o laço para a próxima interação  
**Break –** Para e sai do laço.

Os dois pode ser utilizado no If, For e While.

**Try, Catch e Throw**Try - Bloco onde o código que pode gerar uma exceção é colocado.  
Catch - Bloco que captura e trata a exceção.  
Throw - Usado para lançar explicitamente uma exceção.  
Ex:



**Finally**O finally é um bloco que sempre será executado e vem depois do catch.

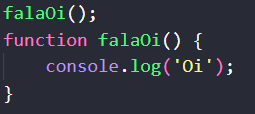
**setInterval e setTimeout**setInterval – Podemos usá-lo para executar um código infinitamente. Podemos também colocar um intervalo que esse código vai ser executado de novo.  
Ex:  
setInterval(function(), tempo)

setTimeout – Usa um intervalo de tempo para exprimir ou executar um bloco de código.

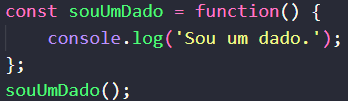
clearInterval – Termina a execução de um código de acordo com o tempo estimado.

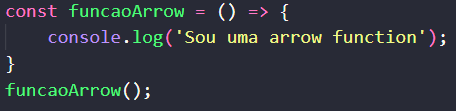
Ex:



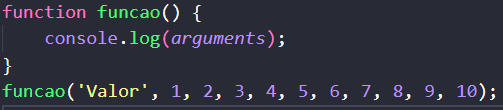
**Funções  
Function Hoisting -**Quando é declarado uma função normal, é possível chamar a função em qualquer lugar do código, sendo antes ou depois da declaração, pois acontece o Function Hoisting.  
Ex:  


As functions são objetos de primeira classe.

**Function Expression -**São as funções que são guardadas em uma variável.  
Ex:  


**Arrow Functions**São basicamente as funtions dentro de uma variável, mas de uma forma mais simplificada.  
Ex:  


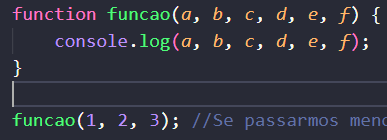
**Parâmetros da Função**Se não passarmos nenhuma variável para a função, mas passarmos parâmetros para ela, eles basicamente ficarão ocultos, só conseguiremos vê-los se passarmos a palavra arguments.  
Ex:

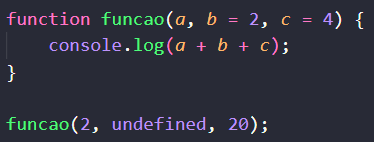


Podemos também escolher qual valor pode ser exibido, só precisamos passar os índices.

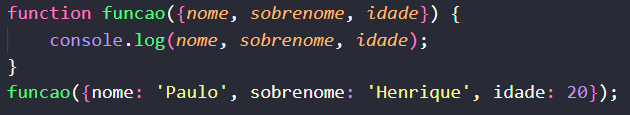


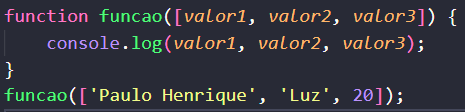
Uma função tem a palavra chave arguments que sustenta todos os argumentos enviados.

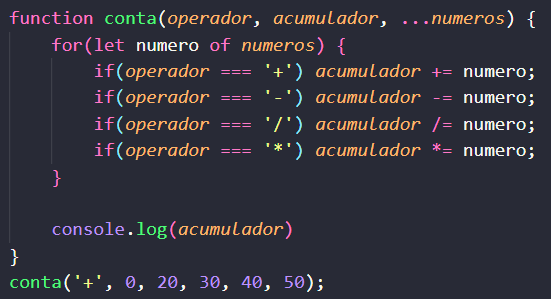
Se passarmos criarmos várias variáveis em uma função e passarmos menos valores do que o necessário, o resto dos valores que faltaram, serão undefined.  
Ex:  


Se tiver um parâmetro e ele estiver no meio e seja preciso que ele assuma um valor padrão, é preciso utilizar o undefined.  
Ex:  


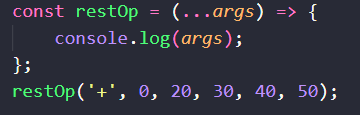
Pois se colocarmos null, ele será contado como 0. E se colocarmos uma string vazia, será contado como concatenação.

**Função com desconstrução**Devemos passar os valores da referência, pois é uma função que usa a desconstrução de objeto.Ex:  


**Função com um Array**Ex:  


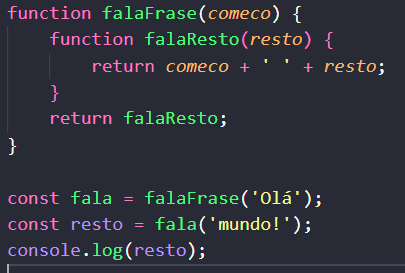
Podemos pegar os valores restante usando o rest operator, mas só é possível no último valor. Quando usamos o rest, não precisamos colocar algum array, pois ele já é transformado.  
Ex:  


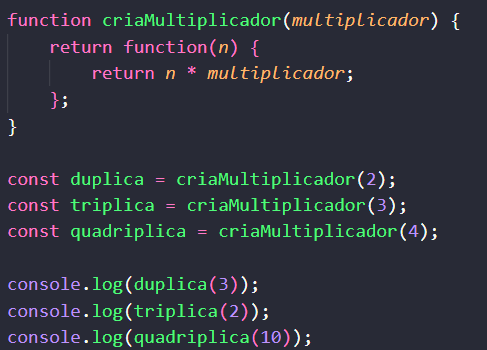
Os últimos valores serão utilizados como um array pelo rest.

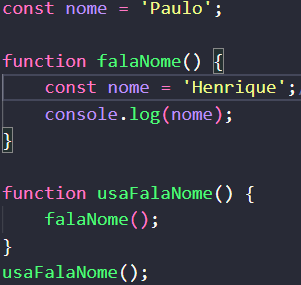
Como não é possível usar o arguments em uma arrow functions, podemos utilizar dessa forma.  


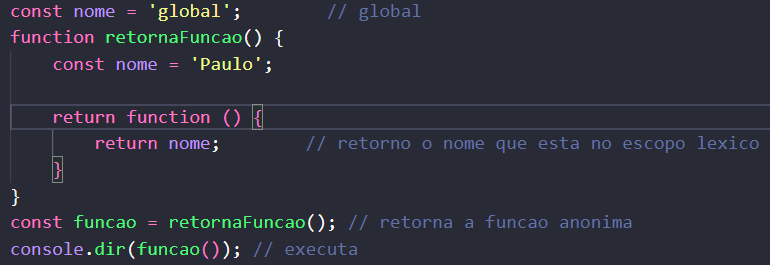
**Retorno da função**O return é utilizado para retornar algum valor, independentemente do tipo.

Podemos também utilizar uma função dentro de uma função. A função menor pode usar os parâmetros da função maior.

Para acessar a função menor, podemos criar uma variável com a função maior e logo depois podemos utilizar pegar a variável e chamar a função novamente, pode ficar confuso, mas aí vai um exemplo.  
Ex:  


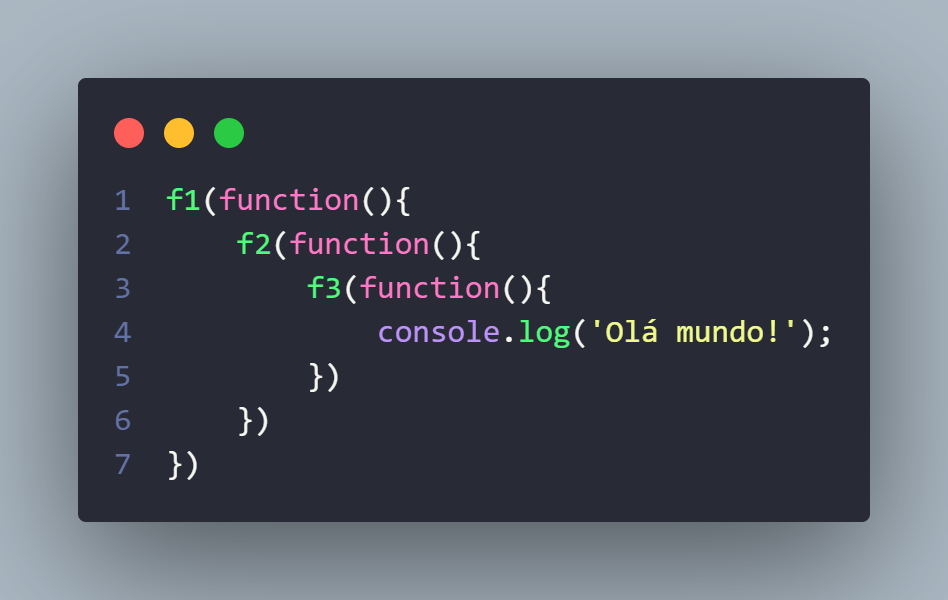
Podemos também retornar uma função dento de um return.  
Ex:  


**Escopo léxico**A função sempre vai conhecer seus vizinhos, ou seja, a função sempre vai procurar o elemento no seu escopo, caso não ache, será procurado no escopo pai e assim sucessivamente.  
Ex:  
  
Se essa variável não existisse, seria puxado a variável do escopo global.

**Closures**É a habilidade que a função tem em acessar o seu escopo léxico.  
Ex:  


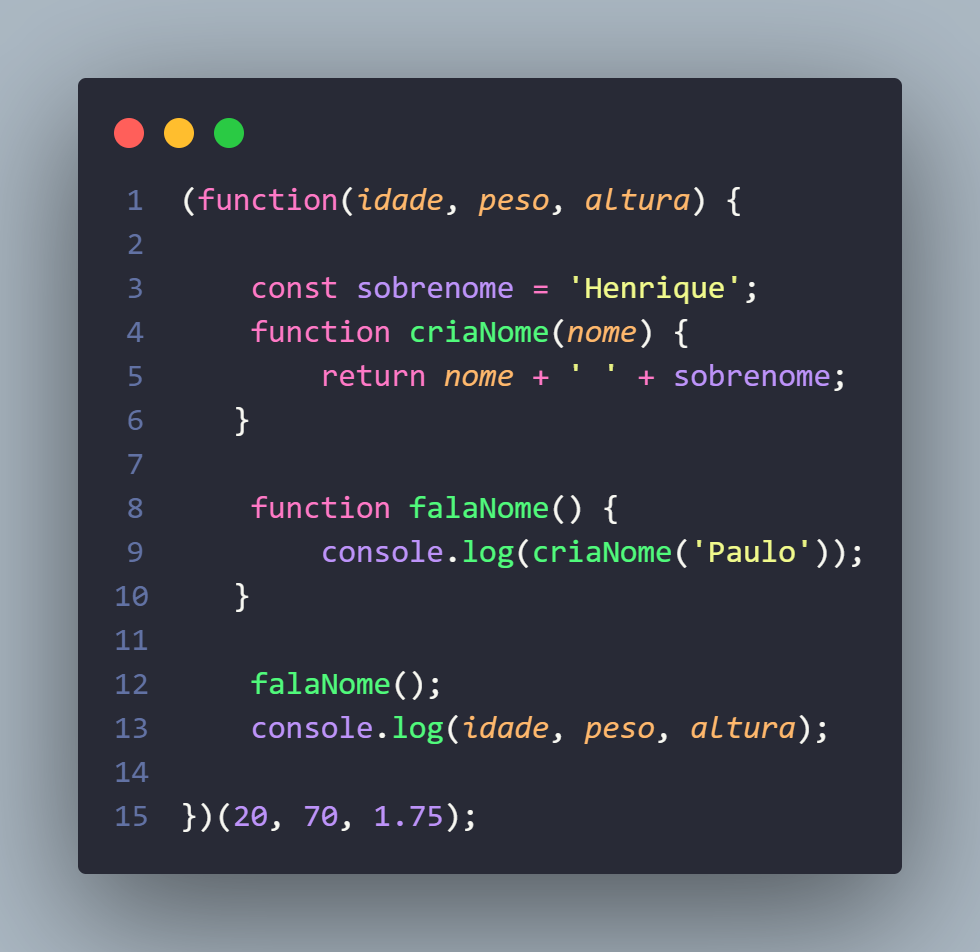
**Funções CallBack**Resumindo é uma forma de controlar a execução de um código para que nada seja executado antes ou depois de uma ordem, ou seja, é chamado em ordem para fazer a melhor execução do sistema.  
Ex:



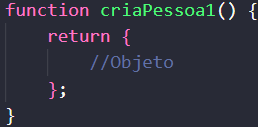


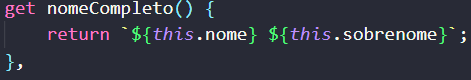
Podemos usar essa forma de encadeamento também.

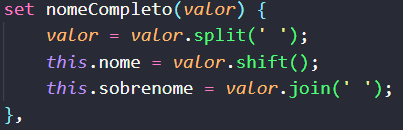
**IIFE -> Imediately invoked function expression.**São funções auto invocadas, ou seja, essa função irá ser executada imediatamente e também ela não possui nome, ou seja, é uma função anônima.  
Ex:

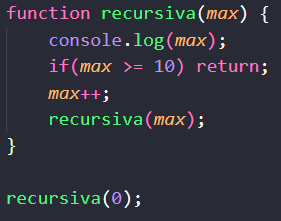


Ela é envolta por parênteses. É possível passar parâmetros no inicio da função, e é possível passar valores que serão lidos pelos parâmetros ao final da função. Sendo assim tudo que é criado dentro, será de seu próprio escopo, fugindo assim do escopo global.

**Factory function**Podemos criar uma função normal que retorna um objeto. É só criarmos uma função e fazer um return com chaves.  
Ex:  


Podemos usar também os getters e setters.  
**Getter**: É uma função que retorna o valor de uma propriedade. É usada quando você quer controlar a forma como o valor é acessado.  
Ex:  
  
Utilizamos o get no começo para dizer que é uma getter. Quando fomos requisitar essa função, não precisaremos utilizar os parênteses.  

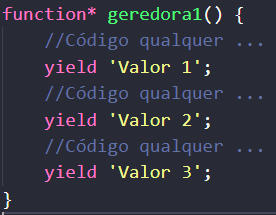

**Setter**: É uma função que define o valor de uma propriedade. É usada quando você quer controlar a forma como o valor é alterado.  
Ex:  
  
Utilizamos o set na frente para se referir que é um setter. É usado igual o getter, não precisamos utilizar parênteses. 

**Funções Recursivas**São aquelas funções que chamam ela mesma, mas dependendo de quantas vezes forem chamadas, possa ser que a engine do js pare com a execução do código, pois pode parecer um erro ou um loop infinito.  
Ex:  


Essa função irá parar, quando o valor chegar no 10.

**Função Geradora**Uma função geradora é uma função especial que pode pausar sua execução e retoma-la posteriormente. Sendo assim é possível retornar valores a cada execução.

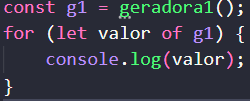
Para criar uma função geradora, é preciso utilizarmos o \* depois da palavra function.  
Ex:  

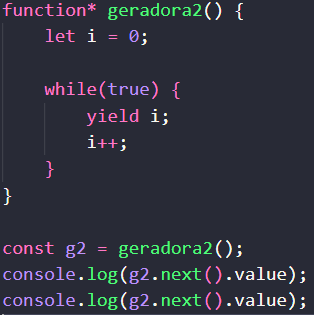

E para retornar os valores, não utilizamos o return, pois se eles forem utilizados, o código é fechado. Então utilizamos a palavra yield.  
Ex:  


Podemos exprimir cada valor de uma função geradora utilizando uma função chamada next(), que é nativa do Js.  
Ex:  

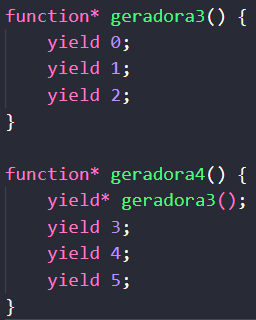

Desse jeito não irá pegar o valor em si, mas sim o yield que está sendo executado, pois para pegar o valor, precisamos utilizar o .value.  
Ex:  


A cada um que passar, será um yield.

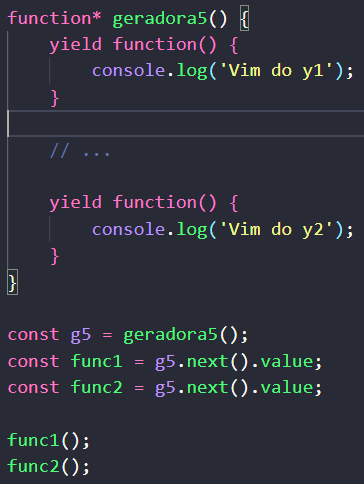
Podemos também utilizar o for para exibir cada valor. Pois o for vai saber exatamente aonde ele deve parar.  
Ex:  


Também podemos fazer um gerador infinito, que percorre cada número a cada laço.  
Ex:  


A cada console.log(), será exprimido um número.

Podemos também utilizar uma função dentro da outra, sendo assim, podemos dar continuidade da outra, para utilizar outra função geradora, precisamos utilizar o “yield\*” desta forma.  


Desse jeito só precisamos utilizar o a geradora4, pois ela já está interando a 4.

Podemos também utilizar uma function normal dentro do yield e depois exprimi-la.  
Ex:  


**Arrays**.splice() – É uma função que possui vária funcionalidades, com ela podemos utilizar o pop, shift, push e o unshift. E ela ainda terá sua própria funcionalidade.  
Ex:

.splice(índice, delete, elem1, elem2, elem3...); - Essa é a forma de como essa função se comporta.

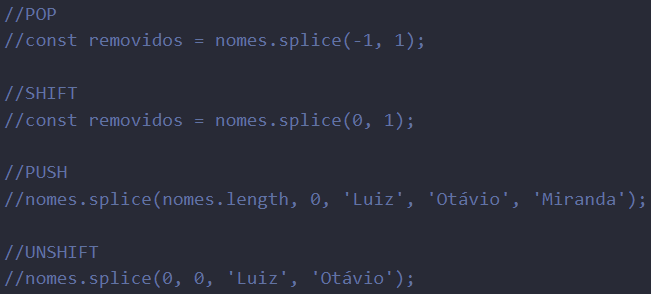
O índice é qual elemento você está “Selecionando”, já o delete é quantos elementos após ele será deletado. Logo após é os novos elementos que podem ser adicionados.

Lembrando que o índice pode ser trabalhado com positivos e negativos, já o delete é utilizado com positivos.

O splice pode ser armazenado em uma variável, ou seja, será exprimido o valor removido do array.

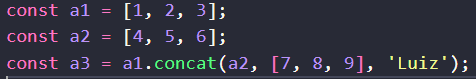
Exemplo de um splice sendo utilizado:  


Logo após o terceiro elemento, será removido dois elementos que será ‘Gabriel’ e ‘Júlia’, e será adicionado o ‘Luiz’ e o ‘Otávio’.

Exemplo de utilização das outras funcionalidades:  


**Concatenação de Arrays**

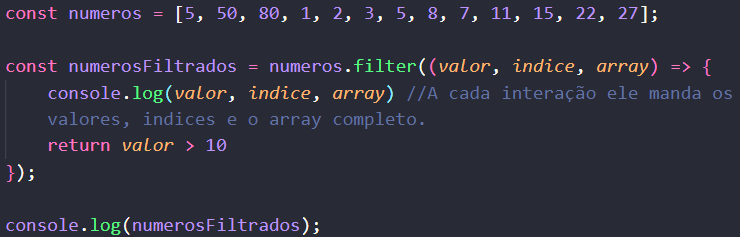
Para fazer a concatenação de array, podemos utilizar dois métodos, sendo ele uma função já pronta do js, ou criando um array e utilizando o rest/spread operator.

Para fazer utilizando a função nativa do JS, precisamos utilizar o método concat()  
Ex:  


Podemos passar outros array, strings ou arrays literários.

Já com o método do rest/spread operator, precisamos criar um array e passar os parâmetros, aqueles que forem arrays, precisamos utilizar os três pontos.  
Ex:  

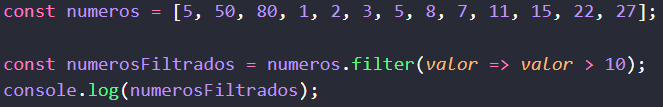

**Filter**Como o nome já diz, é basicamente um filtro que retorna um array, como isso é possível utilizar para diversas coisas e condições.

Ex:  


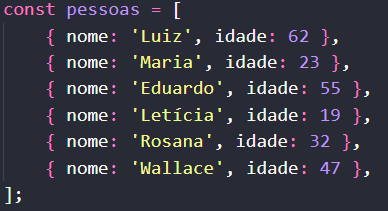
Nesse exemplo precisamos criar um array com os números maiores que 10.

Para usar o filter precisamos criar uma variável e passar o array.filter. Dentro do array é preciso passar uma função, podemos passa-la como função anônima ou arrow function. Essa função pode receber até 3 parâmetros, sendo eles cada valor do array, seu índice e o array inteiro.

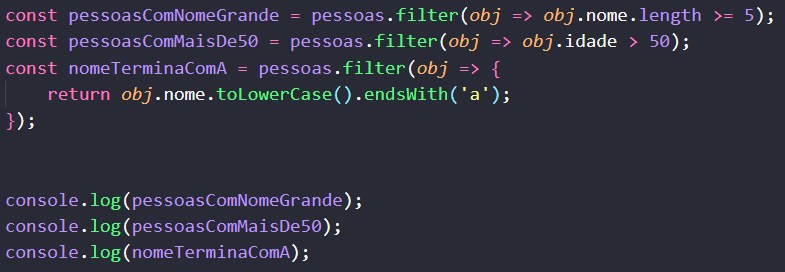
Essa função dentro do filter, apenas retornar true ou false, ou seja, se for true ele será retornado o valor que se encaixa a condição escrita, caso seja false, não será retornado nada.

Aqui também temos a mesma coisa, só que de uma forma mais simplificada:  


Podemos também usar o filter com um objeto, sendo ele usado quase igual com um array, só muda a forma de utilizar o objeto.

Aqui temos um objeto:  


Agora temos que retornar três questões com esse objeto, sendo elas:  
Retorne as pessoas que tem o nome com 5 letras ou mais.  
Retorne as pessoas com mais de 50 anos.  
Retorne as pessoas cujo nome termina com a.

Agora vamos para resolução:  


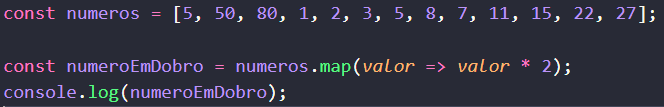
Basicamente está sendo resolvido como foi feito com array, mas dessa vez, para pegar tal linha especifica, precisamos especificar o objeto e pegar a chave especifica. Como está no primeiro exercício.

Se aprofundando mais em uma nova função, podemos utilizar o .endsWith(). Essa função é utilizada para testar se uma string tem determinada letra no final.

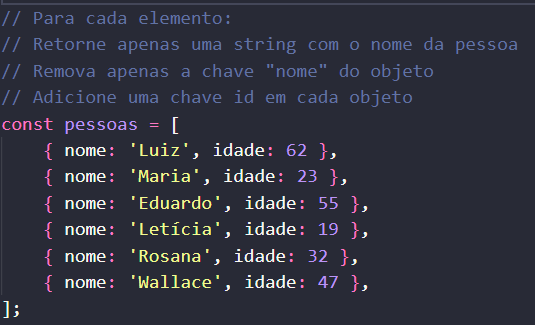
**Map**O método map() em JavaScript é uma função que permite transformar todos os elementos de um array, criando um novo array. Ele aplica uma função a cada elemento do array original e retorna um novo array com os resultados, sem modificar o original.

É quase a mesma coisa do filter, pois pode receber funções de call-back e etc.

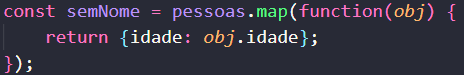
Um exemplo básico com a função map:



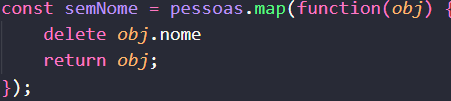
Aqui recebemos um array com número e criamos uma variável que irá receber o map, assim, a cada valor passado do array, será multiplicado por 2 e retornado o novo valor.

Abaixo temos um exercício utilizando objetos:  


Para resolver a primeira questão, será necessário utilizar a forma de selecionar um objeto.  

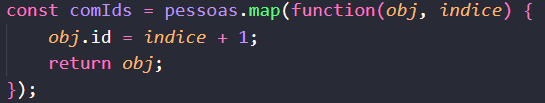

Já na segunda, podemos fazer de várias maneiras.  
Podemos criar um novo objeto e adicionar a chave.  
Ex:  


Aqui estamos criando um objeto e adicionando seus resultados.

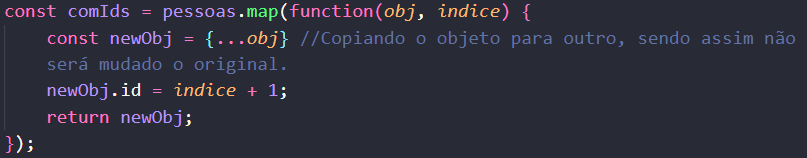
Mas também podemos deletar uma chave do objeto e retornar o resto do objeto.  
Ex:  


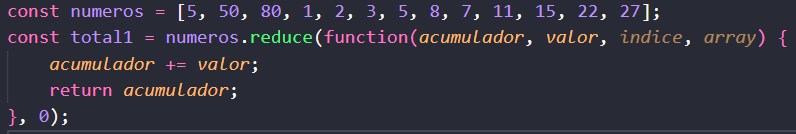
Caso fomos colocar em uma arrow function, temos que ter cuidado nas chaves, pois o compilador interpreta as chaves como se fosse da função e não da criação de um objeto, então devemos colocar parênteses envolvendo o resultado.  
Ex:  


Já na terceira, podemos fazer de duas maneiras, sendo uma que não interfere no objeto original, já a outra que muda o objeto original.

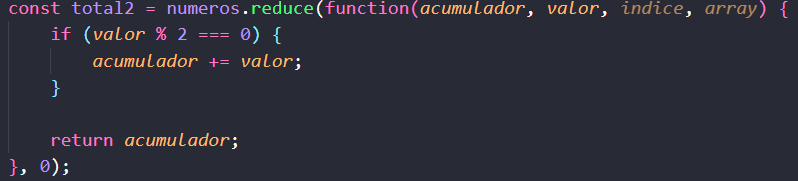
  
Dessa forma estamos criando uma chave id dentro do objeto, e essa chave id irá receber o índice a cada vez que passar por um valor.

O problema é que dessa forma, se exprimimos o objeto original, essa chave de id estará junto.

Mas para mudar isso, precisamos copiar o objeto com o rest operator e adicionar o id no novo objeto.  
Ex:  


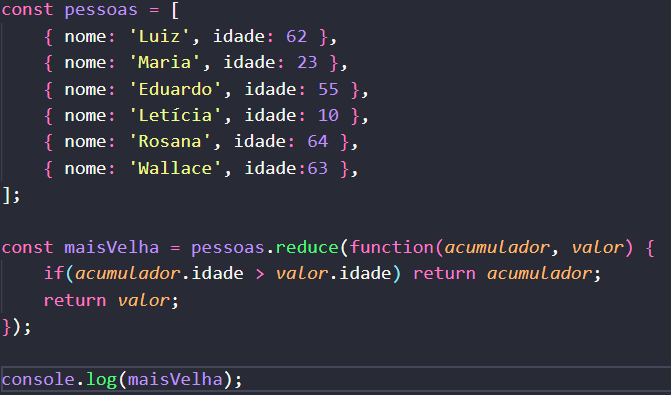
**Reduce**É utilizado para reduzir um array para um elemento. No reduce recebemos 4 parâmetros dentro da função, sendo eles, acumulador, valor, índice e array. Também podemos receber um valor padrão de um acumulador, no caso, o seu valor inicial.  
Ex:  


Caso esse valor inicial não seja mandado, ele será substituído pelo primeiro valor do array.



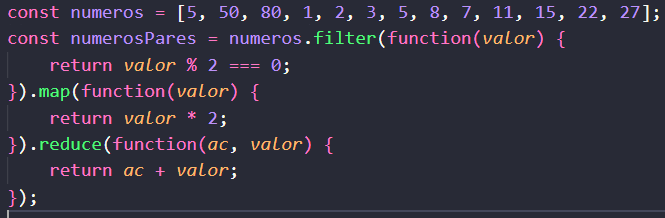
Aqui temos um exemplo que irá somar todos os números que são pares.

Podemos também utilizar em objetos. Iremos fazer um exemplo que devemos pegar apenas a pessoa com a idade mais velha.

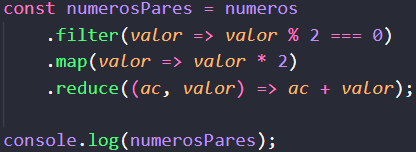


No if, o acumulador está pegando o primeiro objeto, enquanto o valor está pegando o segundo. Ao rodar da aplicação, será testado se o acumulador é maior que o valor. O valor sempre irá atualizar ao passar do código, já o acumulador sempre irá atualizar quando o seu valor for maior que o do próprio valor.

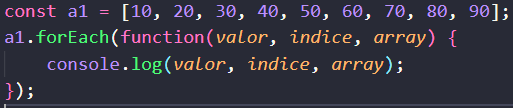
**Filter + Map + Reduce (Exercício)**Aqui vai um exercício utilizando o filter, map e reduce em uma única variável.

A questão é:  
Retorne a soma do dobro de todos os pares  
-> Filtrar pares  
-> Dobrar os valores  
-> Reduzir (somar tudo)  
  


Essa é a forma normal de resolver, logo após terminar cada função, podemos iniciar o outro método.

Já essa é a forma utilizando Arrow Functions  


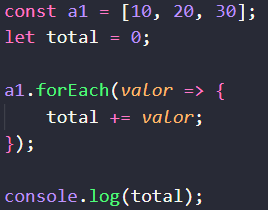
**ForEach**O forEach é utilizado para acessar array, é basicamente o for normal, mas utilizando os arrays.

Como podemos utilizar o forEach:  


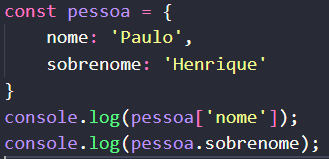
Podemos receber os 3 parâmetros dentro da função, sendo eles o valor, indice e o array completo.



Podemos também utilizar em forma de arrow function, assim o código fica mais limpo.

Podemos também utilizar uma forma de reduce com o forEach, mas é mais aconselhável utilizar o próprio reduce.  
Ex:  


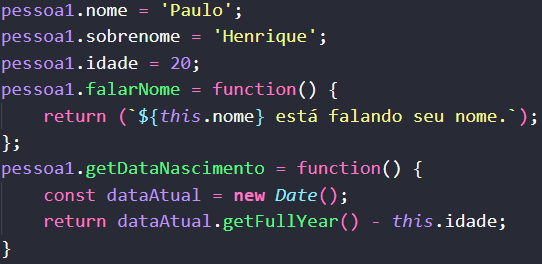
**Mais sobre objetos**Podemos utilizar um objeto de duas formas, sendo elas:

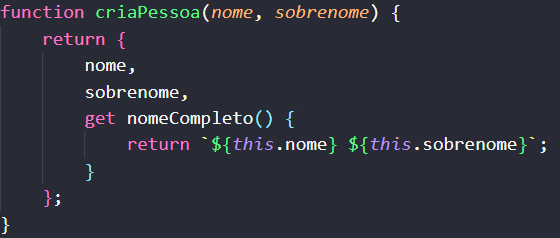


Existe a notação por ponto e por colchetes.  
A notação por colchetes é mais verbal, mas as duas funcionam do mesmo jeito.

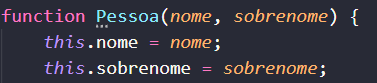
Podemos também utilizar de forma dinâmica, recebendo o valor de um banco ou de algum input para ser armazenado em uma variável.  


Podemos criar um objeto utilizando o new object.  
Ex:  


Dessa forma poderemos adicionar os atributos e métodos fora de um escopo.  
Ex:  


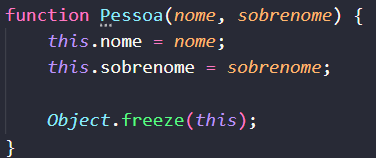
**Factory functions (Objeto)**Podemos utilizar as Factory functions em um objeto, é basicamente uma função normal, mas que irá receber um objeto.  
Ex:  


**Constructor functions (Objeto)**Para criarmos uma constructor function, temos que saber que essa função irá se comportar como um construtor, ou seja, ela terá que ser inicializada como um objeto. Também é necessário ser criada com a primeira letra maiúscula.

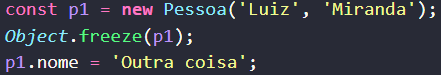


Utilizando a função:  

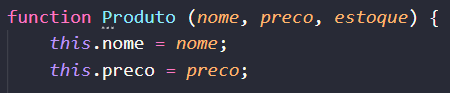

Podemos trocar o valor de um objeto, pois ele está alocado em uma local na memória. Quando alteramos, não estamos trocando o local da memória, mas sim o valor do local da memória.  
Ex:  

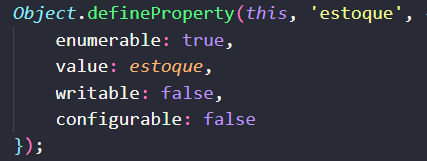

Podemos também utilizar o Object.freeze(). Essa função faz com que determinada variável seja bloqueada, ou seja, não é possível alterar seu valor do local da memória.   
Ex:  


Podemos congelar o this, assim, qualquer coisa que vier depois do freeze não será manipulada, será apenas ignorada. Isso também funciona com arrays.

Também é possível congelar um objeto, então, ele não poderá mais ser alterado.  
Ex:  


**defineProperty e difineProperties**Os defineProperty servem para definir características para variáveis dentro de um objeto. O que muda de um para outro é que, o do singular é utilizado para apenas um item, já o do plural é utilizado para vários.

Função utilizada:  


Para utilizarmos o defineProperty, usaremos essa estrutura:  


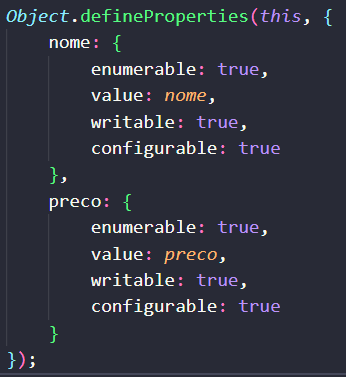
Aqui estamos criando o defineProperty e passando o this, pois ele é uma variável, e logo após passamos o nome da chave.

Cada um desses itens possuem uma característica.  
Enumerable -> Mostra a chave. Se o enumerable estiver false, não será possivel ver pelo object.keys(), ou pelo for in.

Value -> Define o valor Da variável.

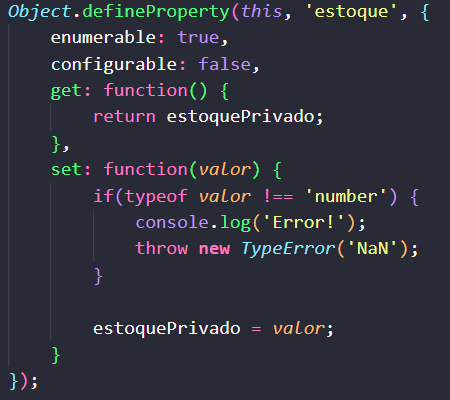
Writable -> Dita se é possivel alterar o valor ou não.

Configurable -> Dita se é possivel reconfigurar, ou seja, criar outro defineProperty ou deletar a variável.

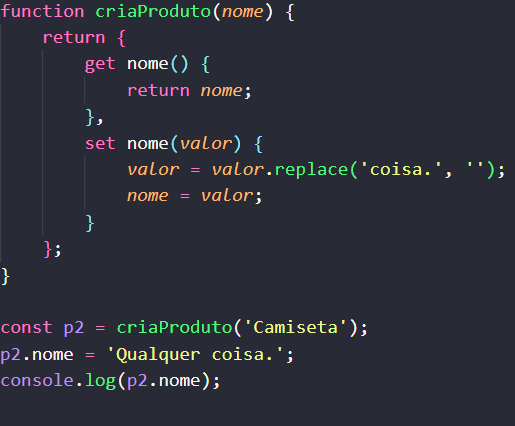
O defineProperties é quase a mesma coisa, pois o que muda é a quantidade de itens que irão ser utilizados.  
Ex:  


Aqui estamos utilizando o nome e o preço, então precisamos remover a variável que foi criada lá em cima. De resto é padrão.

**Object.keys()**O object.keys() é utilizado para mostar as chaves do objeto selecionado.  
Ex:  


**Getters e Setters**Quando utilizando o getter e setter dentro de um defineProperty, não precisamos do value e do writable.  
Ex:  


Aqui temos um exemplo que é, quando o valor for visualizado pelo p1.estoque, ele será mostrado o valor do estoque, mas se for alterar, irá ter um teste para ver se o valor é um number ou não, pois caso não seja, será retornado um erro, mas caso seja valido, será passado pelo if.

E aqui vai um exemplo de como usar em uma Factory function  
Ex:  


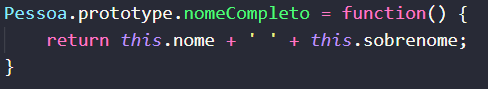
**Métodos úteis para objetos**

Object.assign -> É utilizado para fazer uma cópia de um objeto, sendo assim, não será copiado o local da memória, mas sim criado um novo objeto separadamente. Podemos receber dois parâmetros padrão. O primeiro é um objeto vazio, já o segundo é o objeto cópia e demais objetos.  
Ex:  


*Object*.getOwnPropertyDescriptor -> O getOwnPropertyDescriptor é usado para ver as  
propriedades do objeto, no caso elas são as mesmas que definem o defineProperty. No primeiro parâmetro é preciso receber o objeto, já no segundo é preciso receber a chave do objeto.  
Ex:  


*Object*.values -> É quase a mesma coisa que o Object.keys, só o que muda é que o values irá pegar os valores das chaves.  
Ex:  


*Object*.entries -> É a junção do keys e do entries, pois ele retorna um array com as chaves e os valores do objeto.  
Ex:  


**Prototypes**O prototype é utilizado em funções construtoras em JavaScript para definir métodos e propriedades que são compartilhados por todas as instâncias criadas com essa função. Isso economiza memória e reduz a duplicação de código, pois os métodos definidos no prototype são armazenados uma única vez e utilizados por todas as instâncias. Métodos que não dependem de dados privados e são comuns a todas as instâncias são ideais para serem definidos no prototype. Métodos que dependem de dados específicos de cada instância podem ser definidos diretamente dentro da função construtora.  
Ex:  
****

Um exemplo de como criar um prototype.

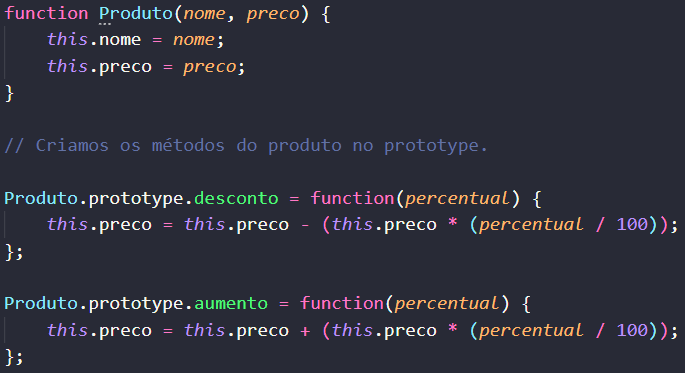
Quando você acessa uma propriedade ou método, o JavaScript segue a cadeia de protótipos até encontrar a propriedade ou método solicitado. Se a busca chega até a função construtora sem encontrar a propriedade, o resultado será undefined.

**Manipulando Prototypes**Podemos utilizar objetos que estão dentro dos outros. Caso tenha o objeto A e ele contem os atributos do objeto B, logo se criarmos o objeto C e conter dentro do B, ele terá as propriedades do objeto A.

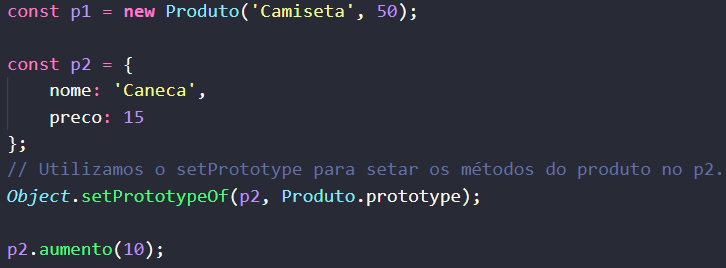
Ex:  


O setPrototypeOf recebe dois atributos, sendo eles, qual será contido e qual contem.

Aqui vai um exemplo melhor.



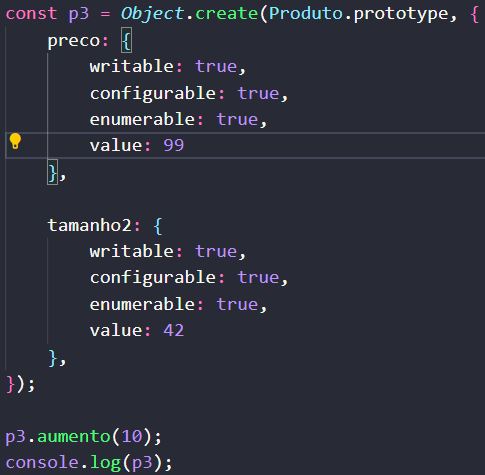
Temos um produto que tem seus métodos criados em um prototype.



Depois criamos um objeto p1, que recebe valores, e logo após criamos o p2 contendo o método do prototype.

Depois utilizamos o setPrototype para fazer com que o p2 receba os métodos que estão no prototype do produto.

E assim podemos utilizar os métodos de um objeto em outro.

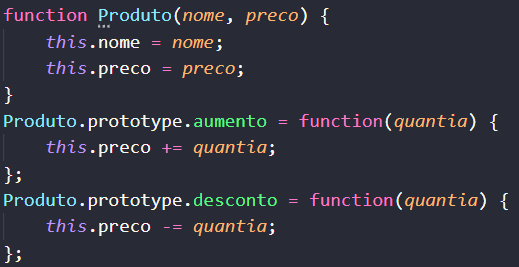
Temos outra forma de criar um objeto e passar direto o prototype de outro objeto.  
Ex:  


Aqui estamos criando o objeto com o Object.create, e no primeiro parâmetro, estamos passando o prototype do produto. Logo após, estamos criando um objeto e passando os atributos. Nesses atributos, podemos utilizar a forma do defineProperties.

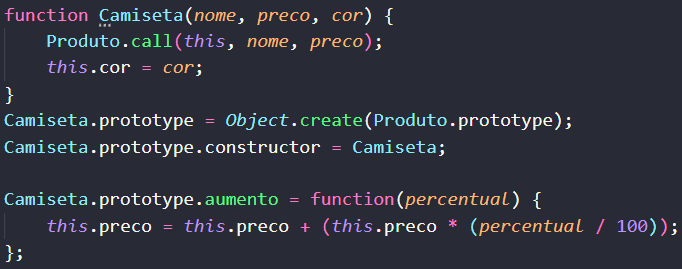
**Herança**O termo "**Herança**" é muito usado para descrever que as características de um objeto são passadas para outro objeto. Assim como uma filha ou filho pode herdar características da sua mãe ou pai (geneticamente). Por isso estamos usando esse termo para descrever o comportamento desejado.

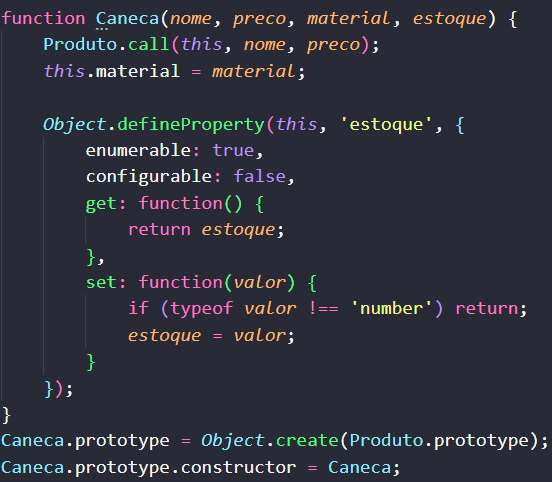
Javascript não é uma linguagem que apresenta essa característica em específico, portanto não existe "**herança**" aqui. O que ocorre, na verdade, é "**delegação**". Esse seria o termo mais correto para descrever "**Herança**" em JS.

Temos um “sistema” que contem Camisetas e Canecas, então, podemos encaixar esses dois em produtos.



Aqui temos o produto que é a função central.





Já aqui temos as outras duas funções que Herdam as características de produto.

Para fazer com que as outras funções herdem de produto, utilizamos a função call, que receber o parâmetro this mais o nome e o preco que vem da função produto e se liga da função Camiseta e Caneca.

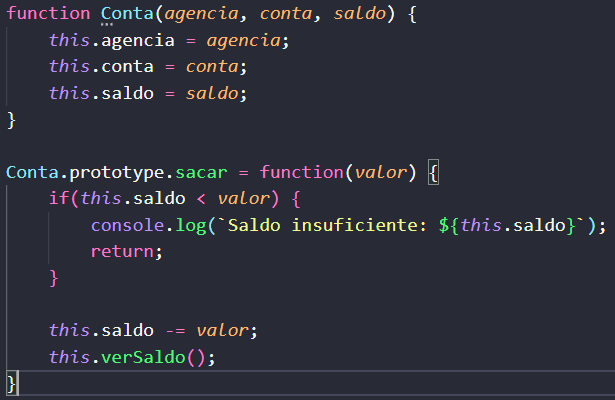
Para podemos utilizar os métodos da função produto, precisamos passar como referencia para criação de um objeto, por isso, usaremos o prototype do objeto, irá receber o Object.create com o parâmetro do Produto.prototype.

Mesmo fazendo isso, ainda precisamos fazer mais uma coisa, pois quando exprimimos essa função, seu construtor é oculto e mostra como se estivesse utilizando o produto, então, o Objeto desejado deve receber o .prototype.constructor e receber o próprio objeto selecionado.

Se quisermos criar ou modificar um método, é só criarmos como novo, pois se ele já existir no método pai, ele será utilizado o método do seu próprio “escopo”. Caso não seja achado, será usado o método do seu pai.

**Polimorfismo**O polimorfismo é a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras.

Basicamente se criarmos um método em uma classe e sobrescrevermos ele em outra, mas mudando algumas coisas, pode ser considerado um polimorfismo.  
Ex:



Aqui temos uma função conta, que contém um método de sacar, nesse método podemos criar outras funções que utilizam ele e que pode alterar a sua função.

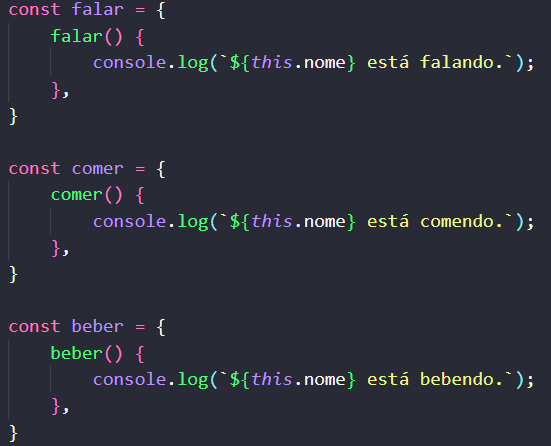


Na conta corrente mesmo, é utilizada a função sacar, mas com uma alteração, pois assim estamos sobrescrevendo a função sacar, mas só para a conta corrente, pois nela possui um limite adicional.

**Factory Funtions + Prototype**Podemos utilizar os prototypes nas Factory functions.  
Temos vários tipos de utilizar os métodos , sendo eles fora da Factory function ou dentro.

Dentro:  


Iremos criar os métodos e colocar dentro uma variável, e depois iremos retornar um object.create, nele iremos passar a variável do prototype e os parâmetros da função, dentro de um objeto. A vantagem de utilizar essa forma, é que podemos definir as propriedades do atributo.

Fora:  


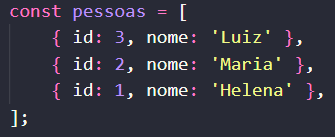
Para utilizar os métodos fora da Factory function, temos que coloca-los dentro de uma variável.  
Podemos ligar os metódos da função com prototype desses dois jeitos, utilizando o assign ou o spread operator



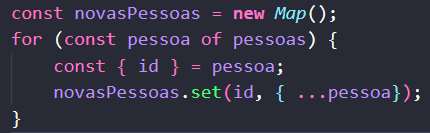
Ou



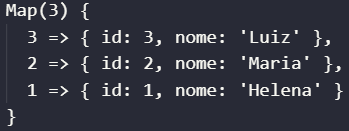
**Map (objeto)**Podemos utilizar o map em um objeto também.  
Ex:

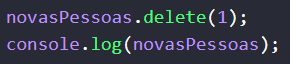


Temos esse objeto e precisamos colocar um indice de acordo com seu array, mas que permaneça na ordem.

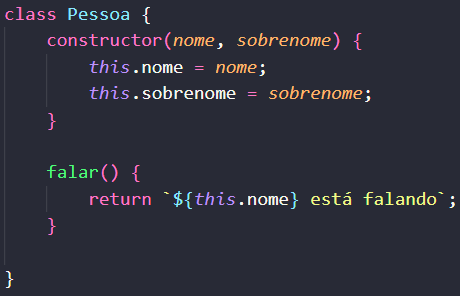


Para isso podemos utilizar o map.  
Instanciamos o map em uma variável. Logo após fazemos um for e desestruturamos a pessoa, pegando apenas o id, logo após pegamos a variável que possui o map e setaremos o valor que queremos.

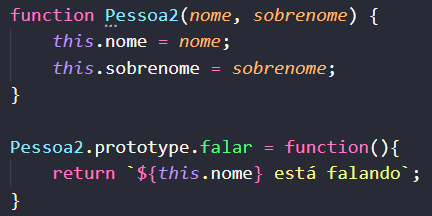
Essa será a resposta:  


Se utilizamos o delete e colocamos algum número de 1 a 3, será deletado o valor que possui o indice  


Nesse será deletado o indice 1, que n caso é o último.

**Criando Classes**

Para criar uma classe, precisamos utilizar class e o nome da classe, esse nome precisa começar com a letra maiúscula. Os parâmetros que a classe irá receber, será armazenado no constructor. Já os métodos vêm logo após. Os métodos não precisam ser colocados no prototype, pois em classes eles já são alocados automaticamente.

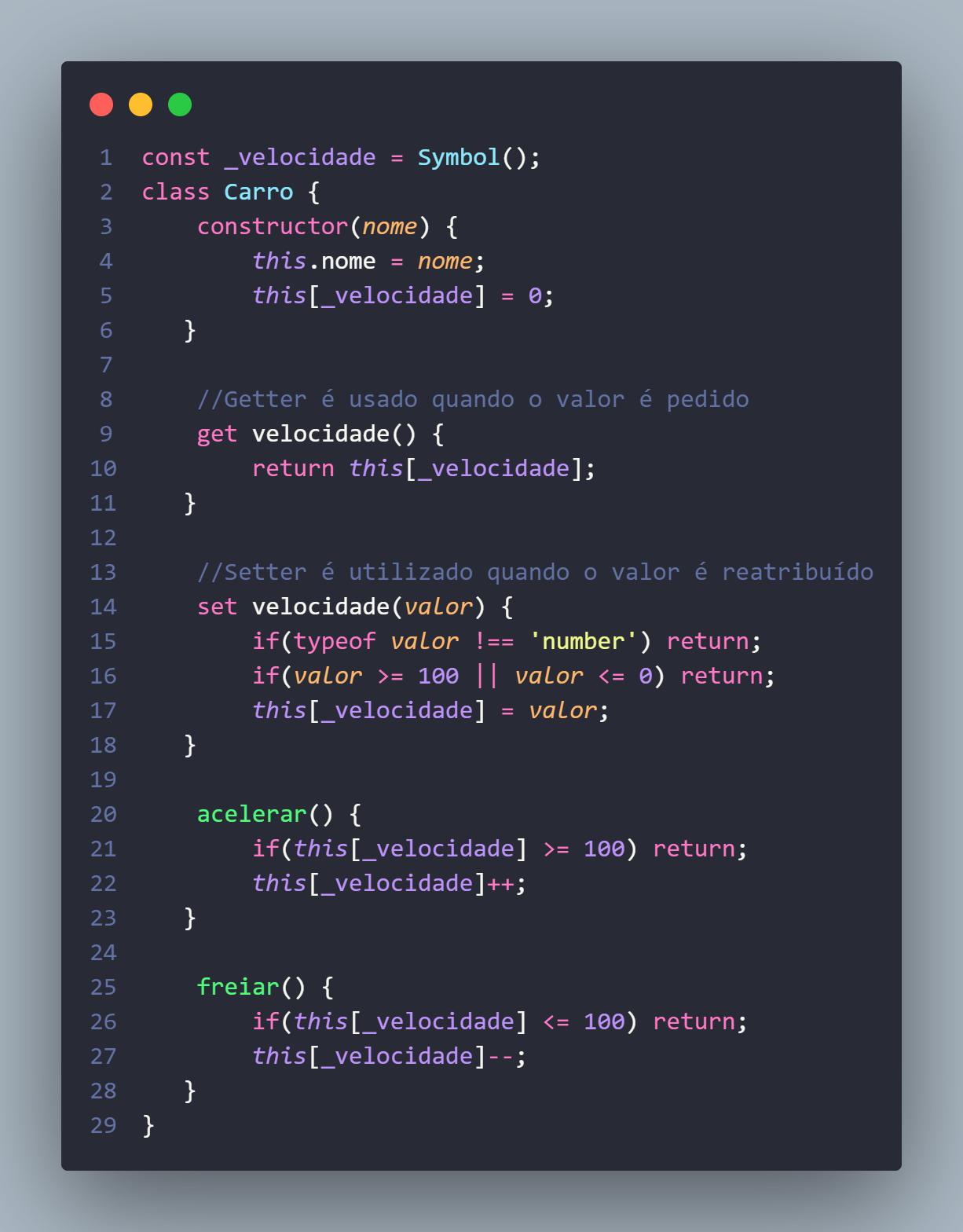
Agora veremos essa classe em funções e objeto.  


**Symbol**O Symbol é um dado primitivo que é usado para criar identificadores únicos e imutáveis, ou seja, um Symbol é único e mesmo que seja criado com a mesma descrição, eles nunca serão iguais

**Getter e Setters (Classes)**A utilização de getters e setter é utilizado igual nas factory e fabric funtions.

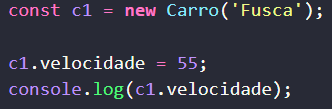
Só para relembrar o Getter é usado quando o valor é pedido ou consultado, já o Setter é utilizado quando o valor é reatribuído.

Agora veremos um exemplo da utilização de getters e setters com Symbol:

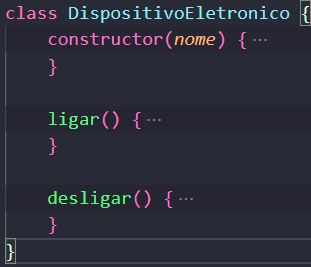


Aqui temos uma classe que possui dois atributos e duas classes.

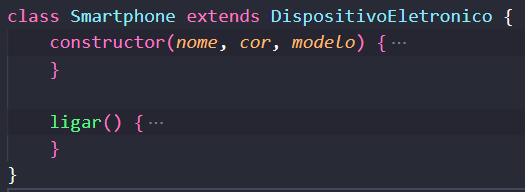
Para que a velocidade não possa ser alterada de qualquer maneira e que possamos valida-la antes de utilizar, utilizamos um novo nome para velocidade e é recebida com Symbol. Dessa maneira qualquer pessoa não poderá settar um valor acima ou abaixo da validação.

Com o getter e o setter podemos utilizar a \_velocidade com outro nome, utilizando apenas um método como getter.  


**Herança com classe**Basicamente a herança de classes é feita pelos filhos, ou seja, todos filhos de uma classe, irão herdar do seu pai, mas os pais não herdarão dos seus filhos.

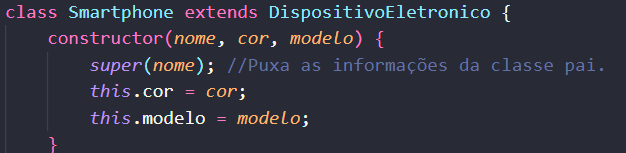
Para fazer com que uma classe herde de outra, precisamos utilizar a palavra extends.  
Ex:  


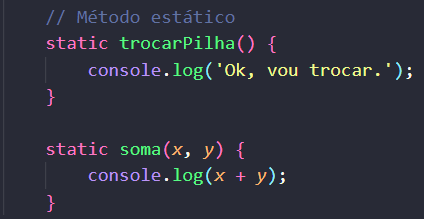
Temos essa classe chamada Dispositivos eletrônicos e precisamos que outra classe herde seus métodos e atributos.



Iremos criar uma classe smartphone que vai se estender de dispositivos eletrônicos.

Para podemos utilizar seus atributos, precisamos utilizar o super(), que irá receber os atributos necessários para estender para outra classe.



**Métodos de Instância Estáticos**Os métodos de instância estáticos é basicamente uma função, mas que não precisa ser instanciada para ser utilizada, pois ela já é instanciada pela própria classe.  
Ex:  


Esses métodos podem receber parâmetros ou não.

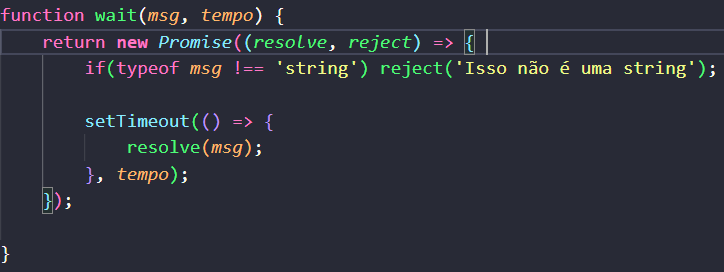


A classe atual se chama controle remoto.

**Promises**Uma Promise é uma forma de executar uma operação assíncrona sem bloquear o resto do código. Enquanto a Promise está esperando por uma resposta, o código continua a rodar normalmente. Se a Promise terminar com um erro, a parte do código que trata esse erro será executada, mas o código após o erro não será executado, a menos que você trate o erro de forma adequada.

Para criar uma Promise, precisamos instancia-la e passar dois parâmetros na sua instancia, que são o resolve e o reject.  
Ex:  

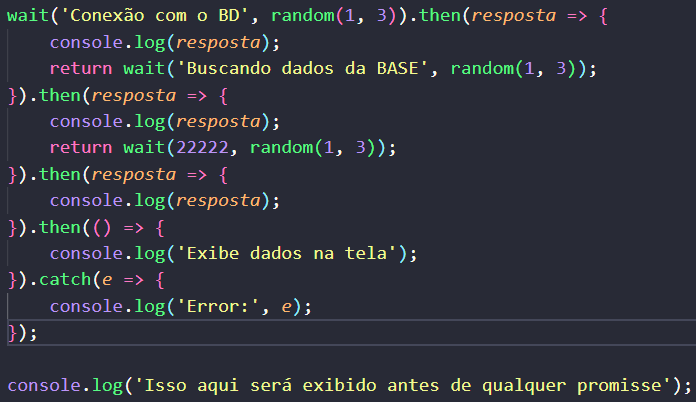

Quando passamos o resolve, ele irá esperar que a operação seja realizada com sucesso, mas quando passamos o reject, é esperado que a operação não seja executada, pois é um erro.

Aqui temos um exemplo de uma função:  


Caso a mensagem receba algo que não é uma string, o código será rejeitado, mas caso seja, o código irá passar.

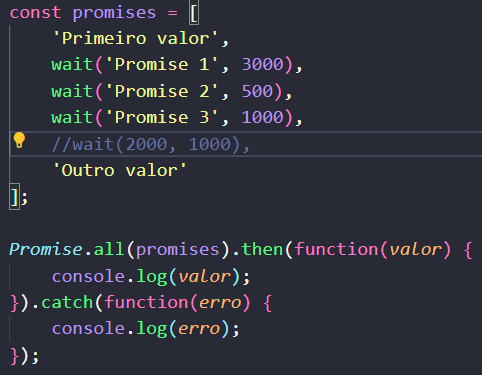
As Promises veem acompanhadas do método then e catch.  
Ao chamar o then, ele irá garantir que determinada função não será chamada em fora de ordem, ou seja, só será chamada, quando a Promise for resolvida.

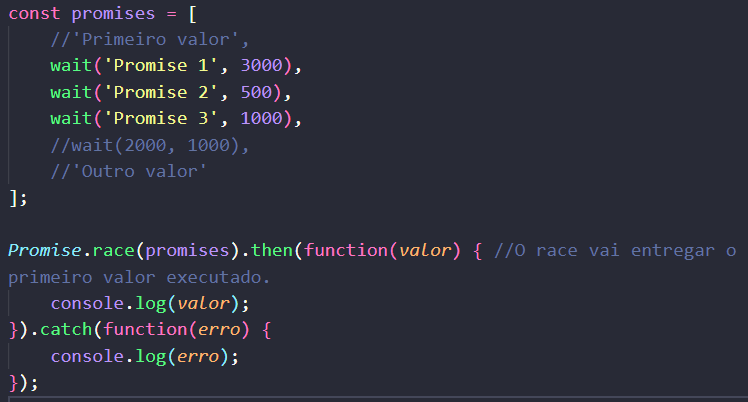
Já o catch irá capturar o erro, caso a Promise não entre em uma condição e seja rejeita.

Ex:  


Quando a execução chegar no inteiro, será quebrada a execução e exprimido o erro.

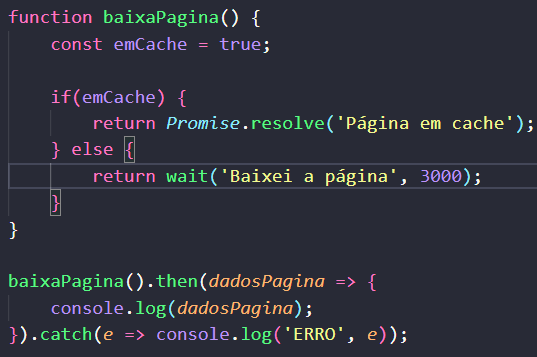
Lembrando que os códigos que vem depois, serão executados.

**Métodos úteis para Promises**Existem alguns métodos que são uteis para utilizar com as Promises, são esses:  
Promise.all - Retorna todos valores, caso não forem recusados.   
Ex:  


Caso tenha algum dado que não seja string, a aplicação irá parar e não será executa, mas caso não possua erros, será esperado até o ultimo dado retornar e será retornado tudo de uma vez.  
Promise.race - Retorna apenas o primeiro valor executado.   
Ex:  


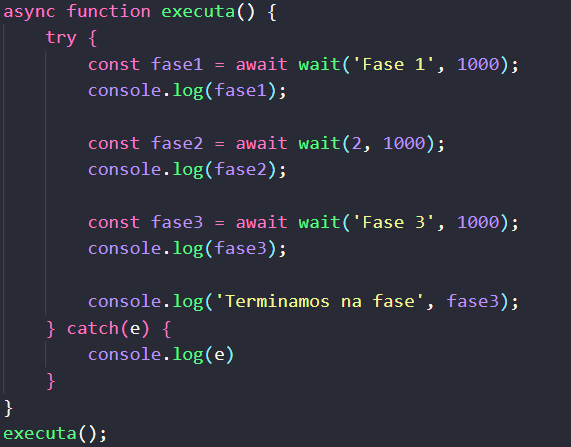
O race irá entregar o valor que primeiro for executado, apenas um.

Promise.resolve - Retorna um valor caso seja aceito.   
Promise.reject - Retorna um valor caso seja recusado.



Basicamente o resolve o reject será executado de acordo com a condição, caso passe por uma condição, será aceito, caso passe por outra, será recusado.

**Async e Await**O async e o await andam juntos com uma Promise, pois a parti deles será retornado um Promise, para dizer que uma função é assíncrona, é preciso colocar na frente dela o async. Já o await é utilizado para que o código seja executado, quando determinado código seja executado.



É basicamente a mesma função do then, só que o async e await foi adicionado em versões mais novas do ecmascript.

Então, enquanto .then() encadeia ações após a Promise ser resolvida, async/await permite escrever o mesmo fluxo de forma mais limpa e parecida com código síncrono.

Eles possuem 3 estados, sendo eles:  
- Pending (Pendente): A Promise está aguardando uma operação assíncrona ser concluída.  
- Fulfilled (Concluída): A operação foi bem-sucedida e a Promise foi resolvida com um valor.  
- Rejected (Rejeitada): A operação falhou e a Promise foi rejeitada com um erro.

**XMLHttpRequest (GET) + Promise**Serve para fazer requisições HTTP de uma forma assíncrona e permite que as páginas da web interajam com os servidores, mas sem recarregar a página, fazendo assim uma atualização dinâmica.

Agora iremos ter um exemplo de carregamento de uma página dentro da outra, só que dinamicamente, ou seja, quando foi clicado no link, será aberta outra página com seus dados.

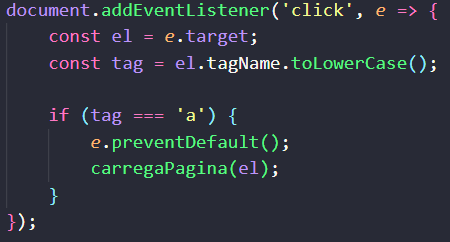


Podemos utilizar uma requisição xhr que retorna uma Promise, para isso precisamos criar uma requisição, que passa um objeto, que vai fazer o retorno da Promise, dentro da Promise, iremos passar dois parâmetros, sendo eles o resolvem e o reject, sendo assim, quando a Promise por rejeitada ou resolvida, será explicito.

Instanciaremos também o xhr e nele iremos passar dois métodos, que é o open, que vai receber o method do objeto, a url e o true.  
O método pode ser GET ou POST, já a url, é a dá api ou dá pagina que irá passar os dados.

Já o send, irá enviar a requisição.

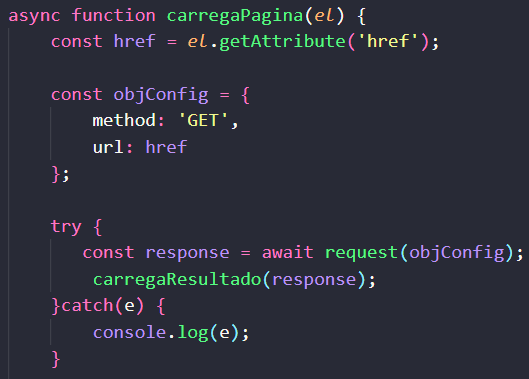
No addEventListener, o load irá fazer com que a requisição seja carregada e depois foi passado as condições para identificar erros ou não.



Evento para carregar o link, só quando for achado uma tag a.

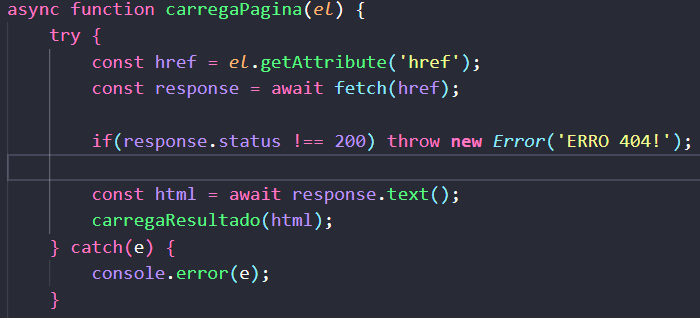
Iremos utilizar uma função assíncrona para que faça o carregamento da página, nela iremos passar o elemento do link, que nele será pego o href do determinado elemento.

Logo após iremos criar um objeto de configuração para requisição, e passaremos o método e o link de referencia que está na variável.

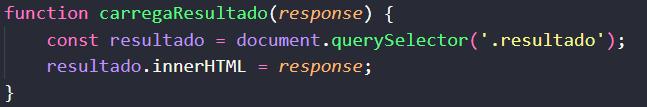


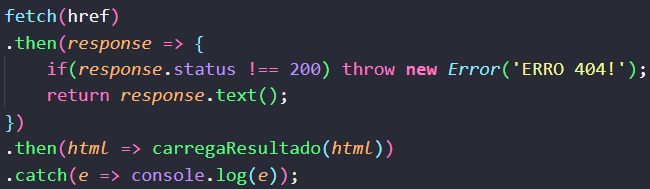
No try, irá aguardar a resolução da função e passar as configurações do objeto para o parâmetro do resultado. Caso apareça um erro, será exprimido no catch.

**Fetch API (GET)**O fetch api é utilizado para fazer requisições de dados de um site ou servidor. Dependendo das informações que forem pedidas, serão respondidas. São retornadas Promises.

Agora temos um exemplo:  


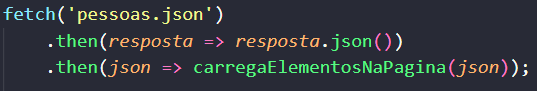
Criamos uma variável response que recebe o fetch com o link da página, que faz esperar até que a requisição seja executada. Se possuir um status de erro diferente de 200, será mandado um erro. Na variável html, irá pegar os dados da página e será transformado em texto. E logo após será executada a função para carregar o resultado.



Podemos também utilizar a forma de Promises  


**Axios**O axios é uma biblioteca que facilita a realização de requisições http, ou seja, é uma forma mais fácil de utilizar o fetch api.

Iremos utilizar um exemplo que iremos receber um json com vários dados de pessoas e iremos exprimir esses dados com axios.

Se fossemos iniciar com fetch, utilizaríamos desse jeito:  


Já com axios, utilizamos apenas isso:  




Na função que carrega os elementos na página, estamos criando uma estrutura de tabela do html e fazendo um for, para que possa pegar o arquivo json e inserir seus dados. Para inserir seus dados, estamos pegando a pessoa do for junto com o nome do parâmetro.

**Babel**O babel é uma ferramenta de transpiling usada para converter o código moderno em uma versão mais antiga, sendo assim, fara com que rode em vários navegadores, independentemente da versão.

Temos um site que podemos utilizar para conversão de código:  
<https://caniuse.com/>

Também podemos usar outro site, que podemos ver em quais versões uma funcionalidade roda ou pode rodar.  
<https://babeljs.io/>

Para instalar as dependências, precisamos utilizar o terminal.

npm init -y

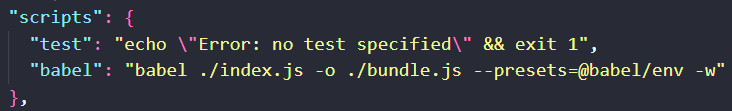
Assim podemos criar o package.json, utilizado para gerenciar os pacotes que irão ser utilizado.

npm install –save-dev @babel/cli @babel/presets-env @babel/core

Este comando faz com que seja instalado as dependências e os pacotes do babel.

npx babel main.js -o bundle.js –presets=@babel/env

Esse comando irá criar o bundle.js na primeira vez e nas próximas execuções ele irá atualizar o arquivo.

Para fazer com que o arquivo seja atualizado sempre que o arquivo for salvo e de uma forma mais rápida, iremos criar uma “variável” dentro do package.json.  


Para executar e deixar o babel assistindo seu código, é só escrever no terminal:  
npm run babel

Que no caso é o nome da “variável” que criamos.

Para fazer que o babel pare de assistir ao código, é só utilizarmos o Ctrl + C

**Webpack**O webpack é uma ferramenta que pode ser utilizada para empacotar os recursos do projeto em um único ou múltiplos arquivos que podem ser carregados no navegador de forma eficiente.  
Site do webpack com documentação: <https://webpack.js.org/>

Para utilizar, iremos iniciar o package normalmente, mas na parte de instalar as dependências do babel, iremos mudar um pouco o código:

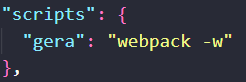
npm install –save-dev @babel/cli @babel/presets-env @babel/core babel-loader webpack webpack-cli regenerator-runtime core-js@2

Caso o repositório seja baixado do github, ou seja, apagado a pasta do node\_modules, podemos instalar novamente, mas sem utilizar o comando gigantescos.  
npm i ou npm install

Depois iremos criar um arquivo chamado webpack.config.js



Depois iremos no package criar uma “variável” para fazer com que o webpack fique assistindo o arquivo.



Depois iremos utilizar o npm run gera, assim ele irá criar o bundle.js e o bundle.js.map

Como fica a arvore das pastas:  
