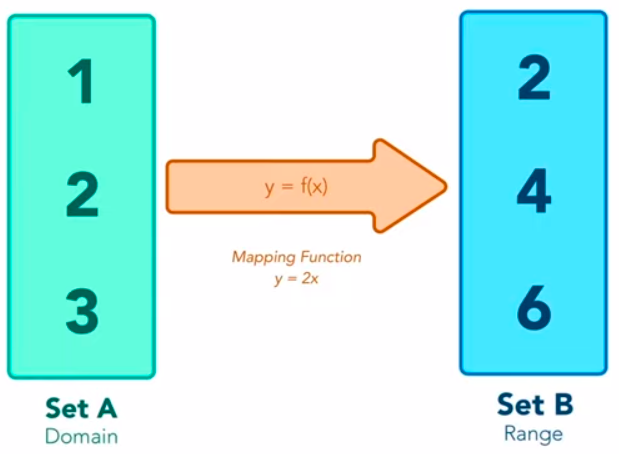
**Map**

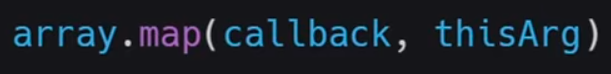
Se pegarmos uma referência matemática, veremos que o **map** é quando um conjunto passa por uma operação se tornando outro conjunto.

Temos a exemplificação disso na figura abaixo. O conjunto A passou pela operação da “mapping function” onde y = 2x, ou seja, o conjunto B é y e o A é x, então o conjunto B receberá os valores de A multiplicados por 2.



E é exatamente isso que a **map** function faz com as arrays, ela efetua operações em cada um dos itens de uma array, sem modificá-la! Os novos valores são armazenados em uma nova array, assim como na figura acima um novo conjunto B foi criado.

A sintaxe de map é bem simples, depois do nome da array, chamo o método map utilizando ponto e dentro dos parênteses colocamos dois itens que podemos receber de volta podendo eles ser o *callback* e o *thisArg* (que é basicamente igual callback mas é opcional):



Usamos **thisArg** somente se for necessário fazer uma operação que vai variar de acordo com o objeto e que essa operação precise ser feita dentro desse objeto. Não é muito usado como argumento, geralmente vemos somente o callback.

**Callback** é a função a ser executada em cada elemento. Dentro dele temos acesso ao item, ao index desse item e ao array original também:



Não é necessário chamar todos parâmetros acima do callback, podemos chamar apenas um, por exemplo.

Map vs forEach

O **forEach** é outro método bem parecido com o **map**, que também faz uma operação em cada elemento da array.

Na figura abaixo temos um exemplo dos dois métodos sendo usados em funções exatamente iguais, utilizando map para multiplicar cada item da array por 2 e obter uma nova array com esses resultados e utilizando forEach da mesma forma, porém o retorno será “undefined”, pois ele não retorna exatamente uma array, seria necessário colocar o resultado desse método dentro de uma constante para poder manipulá-lo, ou receber uma array por ex, como seu resultado:

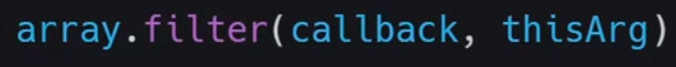


Então basicamente a diferença entre **map** e **forEach** é o resultado, ou seja, o valor de retorno dos métodos é diferente. É preciso sempre considerar se o array auxiliar será necessário na hora de manipular esses métodos.

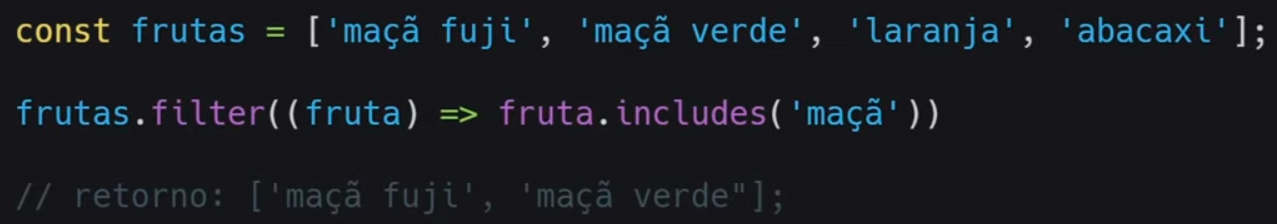
**Filter**

Este método funciona como se colocássemos a array em um filtro e apenas os itens que corresponderem a condição criada serão retornados com resultado através de uma nova array que também é criada com esse método (assim como o map).

A sintaxe dele é igual a do map, recebendo como parâmetros o **callback** (a função que será executada em cada item, sendo geralmente uma comparação) e o **thisArg** (que também é opcional, utilizado somente quando precisamos manipular um objeto específico) :



No exemplo abaixo temos uma lista com várias frutas, duas delas tem o nome maçã. Vou utilizar o filter com a função de callback para me retornar o item (nesse caso nomeei como **fruta**, mas pode ser qualquer nome) e se a fruta inclui o nome “maçã” então, terei os valores destes itens como resultado:



Utilizamos “.includes” pois os itens da array frutas são strings, e includes é um método específico para strings. Caso fizesse esse callback com **frutas === maça** não teria nada como resultado pois nenhuma das strings é somente maçã, tem um segundo nome junto “fuji” e “verde”.

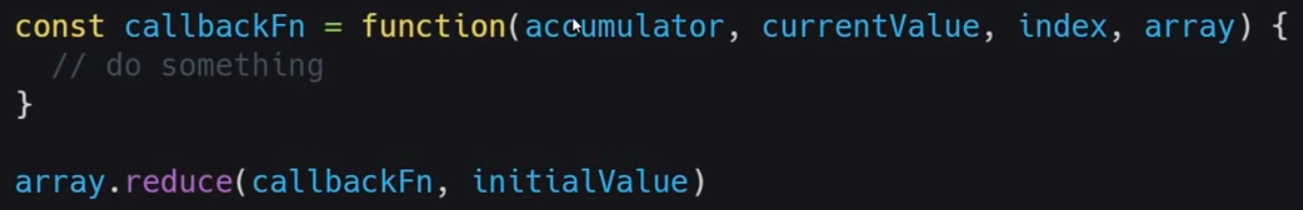
**Reduce**

Diferente do map e do filter o **reduce** não retorna uma array, mas sim um valor único! Então ele executa uma função nos elementos do array e retorna somente um valor.

Ele pode até retornar um array, mas geralmente são uma string, um número, etc.

Sua sintaxe é um pouco diferente alterando somente os parâmetros recebidos nela, ao invés de thisArg agora temos o **initialValue** que também é opcional (ele é o valor sobre o qual o retorno final irá atuar) e o parâmetro **callback** é uma função a ser executada a partir do acumulador que ela possui. Essa função vai verificar o acumulador e executar a função novamente nele.

O exemplo abaixo mostra o que é possível acessar dentro da função **callbackFn**: o acumulador (**accumulator**), o valor atual (**currentValue**), o index do valor atual e o array original onde o reduce na linha abaixo está sendo chamado:



**accumulator/prevValue**: é o acumulador de todas as chamadas de callbackFn.

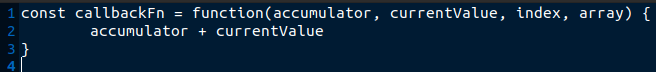
**currentValue:** é o elemento atual sendo acessado pela função.

Não é preciso mandar todos estes parâmetros dentro da função callbackFn, o index e o array são usados somente quando for executar uma função com o index específico etc.

Podemos inicializar com algum valor para o acumulador ou não. Caso não seja passado um valor para **initialValue** o acumulador entenderá que o valor inicial é o primeiro valor da array.

Frequentemente o **reduce** é usado para somar todos os elementos de uma array. Então inicialmente não temos nada, neste caso o initialValue é zero, então não seria necessário passar esse parâmetro pois é zero. O acumulador também seria zero.

Vamos supor que o valor da array seja [1 , 2], então o acumulador é zero, o currentValue na primeira vez que rodar, será 1 (o valor do primeiro item), então supondo que fiz uma função **acumulador + currentValue:**

****

Depois que rodar terei o accumulator = 1, pois o acumulador assume o valor do resultado anterior.

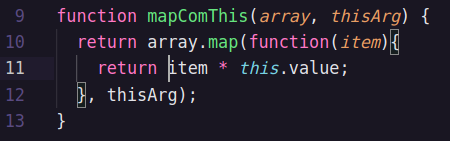
Agora iniciando o segundo ciclo do código, o accumulator =1 e o currentValue = 2 (segundo item da array) e então ele vai somar os dois retornando agora 1+2, ou seja, retorna 3. Então ele somou os valores dessa array!

**Atividade Prática**

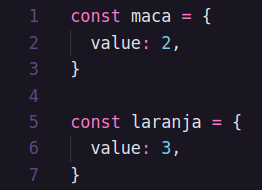
Map

1. Pratique a sintaxe de multiplicação de números, uma vez utilizando o parâmetro this de um objeto criado por você, e depois sem ele.

Vamos fazer primeiro a função map utilizando this, ela retornará um array mapeado (array.map) vamos declarar nosso callback que será function(item), pois quero fazer uma operação com o item da array multiplicando ele pelo value do meu objeto que ainda vamos construir:



Nessa função **mapComThis** posso mandar uma array e depois um objeto que vai ser utilizado como o argumento this (thisArg). Vamos criar nossos objetos para interagirem (como o this) na nossa função:



Se eu mandar maçã como thisArg, ele pega o value de maçã (em this.value), que é 2, e multiplica pelo item da array: **item \* this.value**.

Mesma coisa para a laranja, porém o value é 3 e não 2.

Vamos criar nossa array para interagir na função:



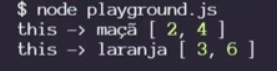
Agora vamos logar a função **mapComThis()**, utilizando a array *numeros* (correspondendo ao parametro array que eu fiz na função) e o thisArg podendo ser maca ou laranja:



Vamos acrescentar um texto só para ficar mais didático e identificar melhor quando cada um dos objetos está sendo chamado:



Se rodarmos o código teremos o resultado:

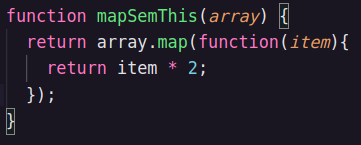


Quando this é maça, o resultado é 2 e 4, pois no array o primeiro item é 1 e o value de maça é 2, então 1 \* 2 = 2. O segundo item da array é 2, então 2 \* 2 (value de maça é sempre 2) = 4.

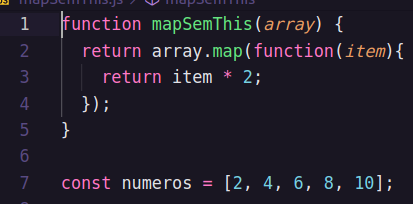
Mesma coisa para a laranja, porém o value será 3.   
Então os itens da array estão interagindo executando a função de multiplicação para cada um dos itens e os values dos objetos maça e laranja.

O this sempre vai estar relacionado ao objeto que eu estiver mandando em thisArg.

Caso não mande nenhum valor para thisArg eu vou apenas chamar a array na minha função sem map, e quando eu receber essa array vou mapeá-la e na minha função *callback* vou ter o item da array e como retorno multiplicarei o item por 2:



Depois crio minha array:



E agora é só logar o resultado da função:



Resultado:

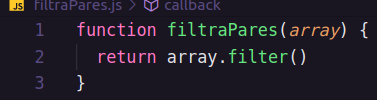


Multiplicou cada um dos itens da array numeros por 2.

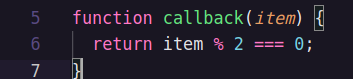
Filter

1. Filtre e retorne todos os números pares de um array.

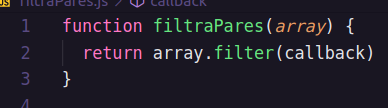
Vamos criar uma funcao “filtraPares” recebendo uma array, vamos ter como retorno um array.filter e a callback function dessa vez vamos fazer do lado de fora e depois a gente manda o array pra dentro dessa função:



Como parametro da nossa callback teremos o item da array, e o retorno será a verificação se é um número par ou não através de **item % 2 === 0:**

****

Com a função callback pronta, agora preciso chamar ela como parametro do meu filter:



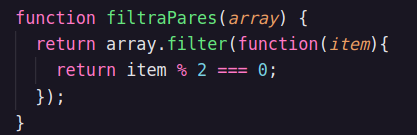
Agora vou declarar minha array:



E por último vou logar minha função:



Poderíamos também ter feito nosso código dessa forma:

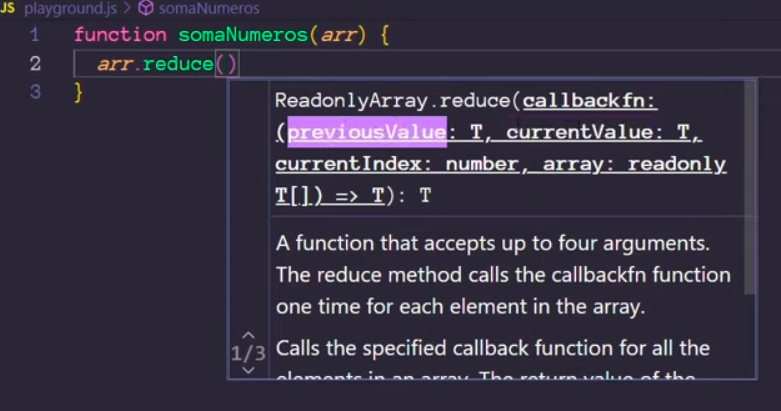


Reduce

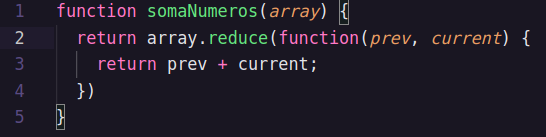
1. Some todos os números de um array (Neste caso temos um valor inicial no segundo não teremos).

Vamos declarar uma função “somaNumeros” recebendo uma array, dando reduce nela.

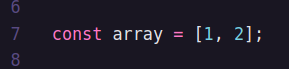
O VScode já mostra pra gnt os parâmetros que podemos passar:



Vamos pegar o valor anterior (prev) e somar com o atual (current):



agora crio minha array:

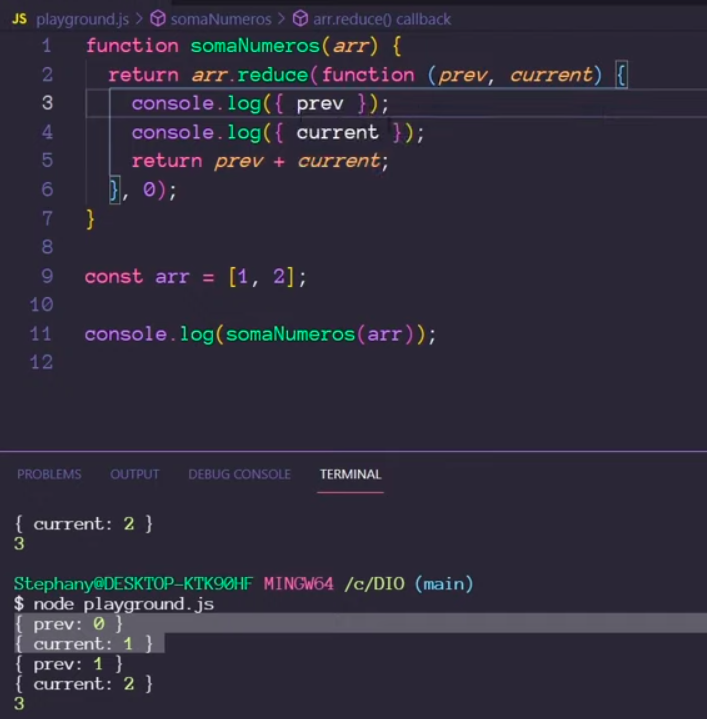


e logo minha função, chamando a array como parametro:



Dessa forma teremos como resultado: 3.

Abaixo temos um exemplo usando console.log para mostrar os valores de prev e current ao longo da iteração pela array para entendermos melhor:



Passando um valor inicial (initialValue) eu devo colocá-lo como segundo parâmetro de reduce. No caso acima foi passado o valor zero como initial value.

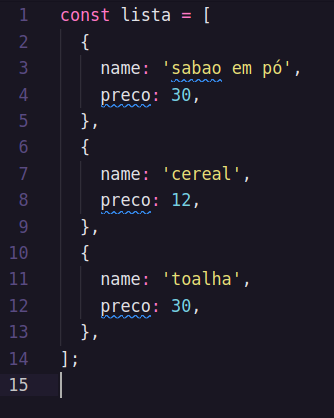
O prev também é chamado de acumuladoor, ele acumula os valores, perceba que ele se tornou a soma na segunda rodada: começou como 0, virou 1 na segunda rodada.

Todo o resultado de cada rodada dessa função de reduce vai definir o próximo valor do previous (prev).

1. Crie uma função que recebe uma lista de preços e um número representando o saldo disponível. Calcule qual será o saldo final após subtrair todos os preços da lista enviada.

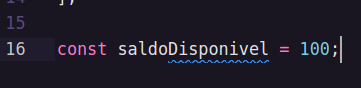
Neste exercício o saldo disponível será o valor inicial e queremos saber quanto de saldo haverá depois que diminuirmos desse saldo disponível o preço de todos os produtos de uma lista.

Vamos declarar primeiro nossa lista de objetos com vários preços:

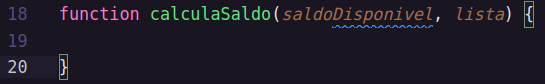


Temos um total de 72 reais em produtos.

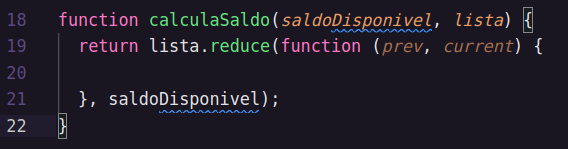
Vamos agora criar nossa constante “saldoDisponivel” dando um valor de 100 para ela:



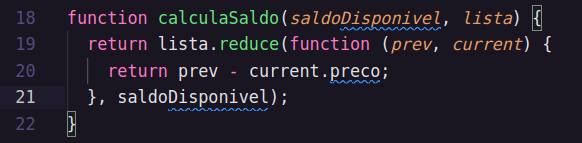
Agora quero saber quanto terei de saldo depois que pagar por todos estes produtos. Então para isso vamos fazer a função “calculaSaldo” que recebe o saldo disponível e a lista dos objetos:

]

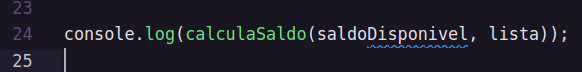
Agora como retorno dessa nossa função vamos usar o reduce no array (que é a lista) usando como argumentos do reduce a função de callback utilizando o **prev** e o **current**. Dessa vez vamos colocar um valor inicial, que será o saldo disponível:



E o retorno dessa callback será a subtração do prev com o current, partindo do saldo disponível. O anterior (prev) vai começar como 100:



Por ultimo vamos chamar a função, recebendo com primeiro parametro o saldo disponivel e o segundo é a lista:

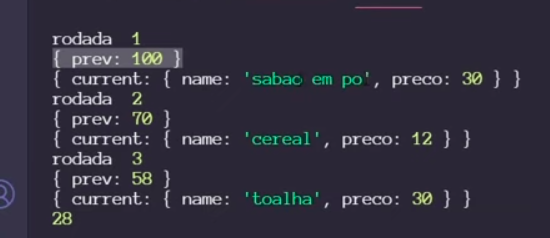


Vamos colocar o console.log para entendermos o que está acontecendo logando também o index para saber qual rodada do código está:



corrigir acima: **return prev - current.preco;**

Resultado:



Rodada 1 o valor anterior é 100, igual ao saldo disponivel. O current é o sabão em pó e o preço é 30. Então 100 - 30 = 70. Então em cima do acumulador (prev) ele faz a operação que é prev - current e assim, iniciando a segunda rodada o prev (acumulador) vira 70.

Na segunda rodada o current é o valor do cereal, ou seja, 12, ele vai pegar o prev que é 70 e subtrair o current 12: 70 - 12 = 58.

Agora o prev é 58 e o current será o valor da toalha 30: 58 - 30 = 28.   
Finalizamos o cód com saldo disponível de 28.