

arrays



array

Consiste em uma **coleção de um ou mais objetos, do mesmo tipo.**

Cada **objeto** é chamado de **elemento do array.**

Estes elementos são **acessíveis individualmente via um índice específico.**

Formas de definição de um arrays em JavaScript:

a) Criação por meio de literais.

```
const alunos = ['Rafael', 'Maria', 'André', 'Ana'];  
console.log(alunos);  
// construção de array literal (mais intuitiva e curta)
```

b) Criação por meio de construtor.

```
const nomes = new Array('Marcelo', 'Denise', 'Hugo');  
console.log(nomes);  
// pode ser aplicado a objetos, arrays, Strings, funções
```

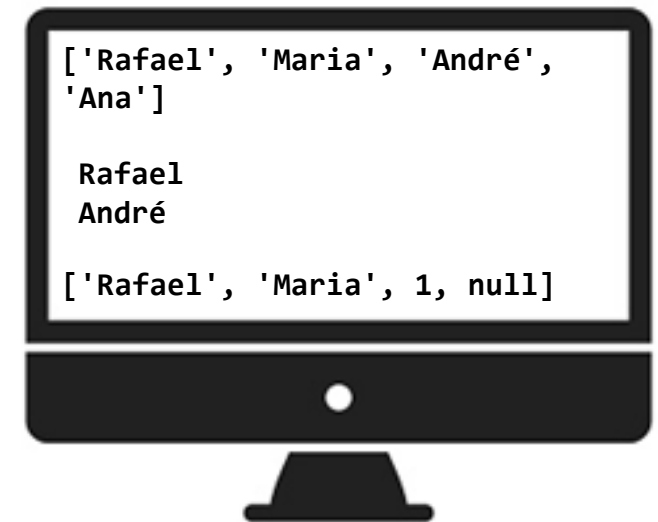
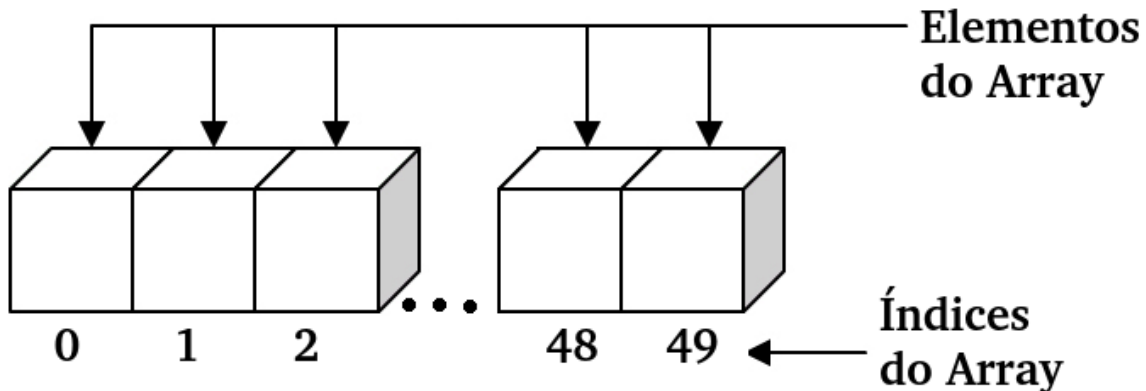




Acessando os elementos nos **arrays**

Os elementos individuais do **array** são acessíveis via **índice**.

```
const alunos = ['Rafael', 'Maria', 'André', 'Ana'];  
console.log(alunos);  
  
console.log(alunos[0]);  
console.log(alunos[2]);
```





Diferentes tipos de dados nos arrays

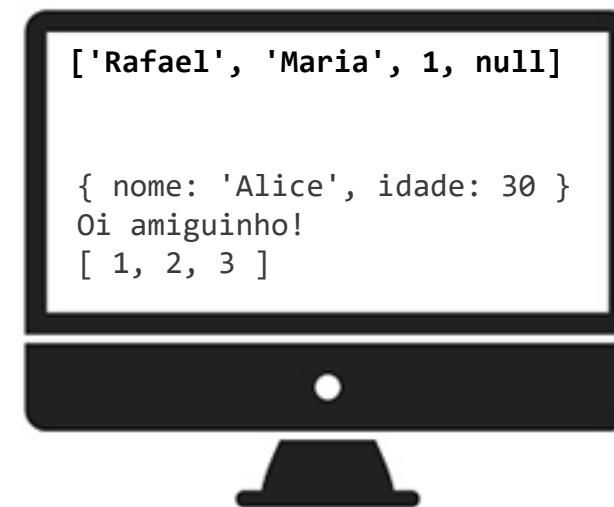
JavaScript permite que um array contenha elementos de tipos variados, como números, strings, objetos, e até mesmo outros arrays.

Isso difere de muitas linguagens de programação que exigem que todos os **elementos de um array sejam do mesmo tipo**.

É possível usar **diferentes tipos de dados em vetor JS**.

```
const alunos1 = ['Rafael', 'Maria', 1, null];  
console.log(alunos1);
```

```
// Array com objetos e funções  
const vetorComplexo = [  
  { nome: 'Alice', idade: 30 },  
  function alo() { return 'Oi amiguinho!'; },  
  [1, 2, 3]  
];  
console.log(vetorComplexo[0]);  
console.log(vetorComplexo[1]());  
console.log(vetorComplexo[2]);
```



No entanto, isso não é uma boa prática.
Melhor manter vetores com elementos de
mesmo tipo de dado.

Operações sobre arrays





Operações sobre arrays

A atualização de um elemento pode ser feita diretamente via **índice no array**.

```
const alunos = ['Rafael', 'Maria', 'André', 'Ana'];  
alunos[0] = 'Eduardo';  
console.log(alunos);
```

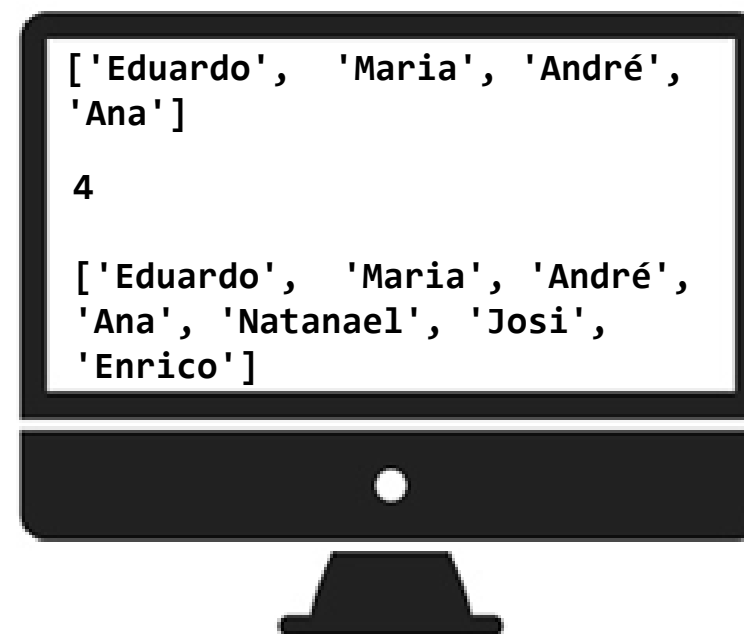
Como descobrir o tamanho do array?

Atributo **length** de um **array**, armazena o seu tamanho.

```
console.log(alunos.length);
```

Uso dessa propriedade é útil para adicionar elementos ao final do array.

```
alunos[alunos.length] = 'Natanael';  
alunos[alunos.length] = 'Josi';  
alunos[alunos.length] = 'Enrico';  
  
console.log(alunos);
```



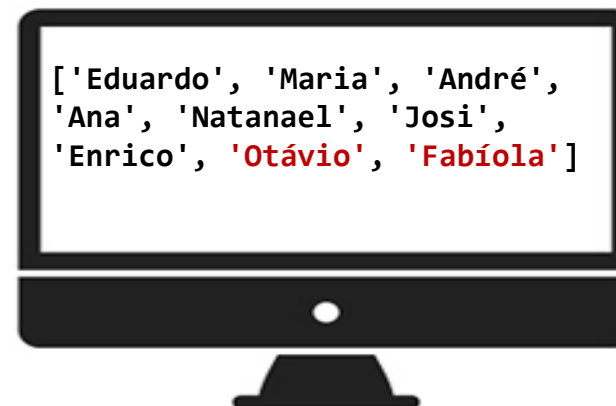


Métodos para manipular arrays

Utilizados para simplificar a manipulação dos arrays.

push()

```
// ['Eduardo', 'Maria', 'André', 'Ana', 'Natanael', 'Josi', 'Enrico']  
// push: insere no final da lista  
alunos.push('Otávio');  
alunos.push('Fabíola');  
console.log(alunos);
```



unshift()

```
// unshift: insere elementos no começo da lista  
alunos.unshift('Jéssica');  
alunos.unshift('Vitor');  
console.log(alunos);
```





pop()

```
// ['Vitor', 'Jéssica', 'Eduardo', 'Maria', 'André', 'Ana',  
'Natanael', 'Josi', 'Enrico', 'Otávio', 'Fabíola']
```

```
// pop() remove elementos do final da lista
```

```
alunos.pop(); // remove Fabíola
```

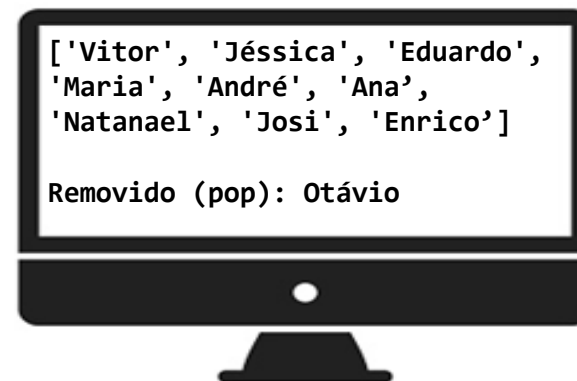
```
console.log(alunos);
```

```
// pop() Permite salvar o elemento removido em uma variável
```

```
const removido = alunos.pop();
```

```
console.log(alunos);
```

```
console.log(`Removido (pop) ${removido}`);
```



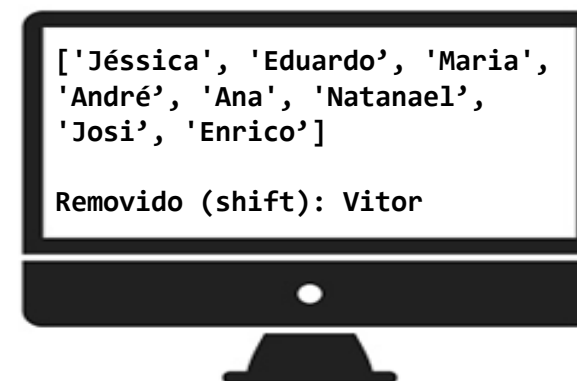
shift()

```
// shift() remove elementos do início da lista
```

```
const removido1 = alunos.shift();
```

```
console.log(alunos);
```

```
console.log(`Removido (shift): ${removido1}`);
```





Operador delete

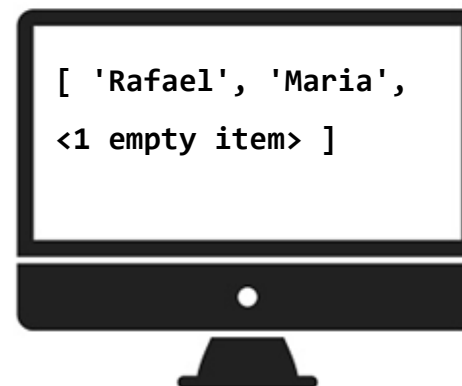
Os métodos vistos até aqui **acabam refletindo sobre os índices**.

Por exemplo, ao inserir um elemento no início da lista (posição 0), o elemento da posição 1 passa a ocupar a posição 2, o elemento da posição 2 ocupará a posição 3, e assim por diante.

O operador **delete** apaga elementos, **mantendo o tamanho original do array**.

O método **deixa um elemento vazio** no lugar do elemento removido.

```
const nomes = ['Rafael', 'Maria', 'Joana'];  
delete nomes[2];  
  
console.log(nomes);
```

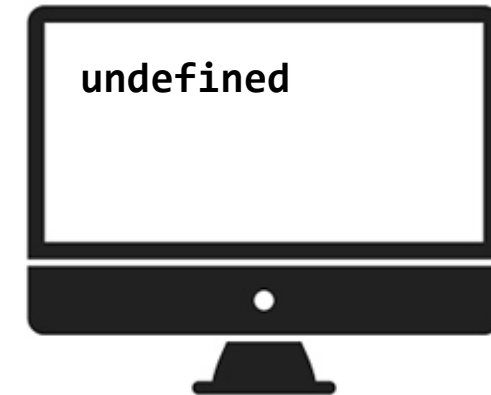




Acessando elementos fora dos limites do **array**

JavaScript permite o acessar índices fora do array.

```
const nomes = ['Rafael', 'Maria', 'Joana'];  
console.log(alunos[20]);
```



Neste caso, o valor **undefined** será apresentado.

Ou seja, **não ocorre um erro**, como poderia ocorrer em outras linguagens, como o C ou Java, por exemplo.

Métodos

slice & splice





Método `slice()`

Permite **recortar (fatiar)** uma parte do array.

```
      0         1         2         3         4         5
const nomes = ['Rafael', 'Maria', 'André', 'Ana', 'José', 'Anita'];
console.log(nomes.slice(0, 3));
```

Recorta os elementos a partir da **posição definida pelo primeiro parâmetro**, até o **elemento anterior ao segundo parâmetro da função**.



```
      0         1         2         3         4         5
const nomes = ['Rafael', 'Maria', 'André', 'Ana', 'José', 'Anita'];
console.log(nomes.slice(1, -2));
```

Recupera os elementos a **partir da posição definida pelo primeiro parâmetro**, e vai até a **quantidade total do array menos a quantidade de elementos** definidas no segundo parâmetro (**quando este valor é negativo**).





splice()

Método curinga que permite realizar diversas operações, inclusive simular as operações feitas com os métodos vistos anteriormente.

Adiciona e/ou remove elementos do *array*.

`array.splice(indice, qtd, item1, ..., itemX)`

Obrigatório: a **posição para adicionar/remover itens**. O valor negativo define a posição do final da *array*.

Opcional: **número de itens a serem removidos**.

Opcional: **novo(s) elemento(s) a serem adicionados**.

Retorno: Um array contendo os itens removidos (se houver).

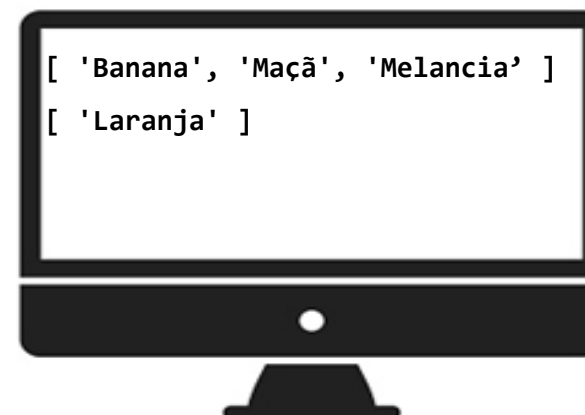
```
const frutas = [ 'Banana', 'Maçã', 'Melancia', 'Laranja' ];
```

```
// simulando pop - remoção no final
```

```
let removidos = frutas.splice(3, 1);
```

```
console.log(frutas, removidos);
```

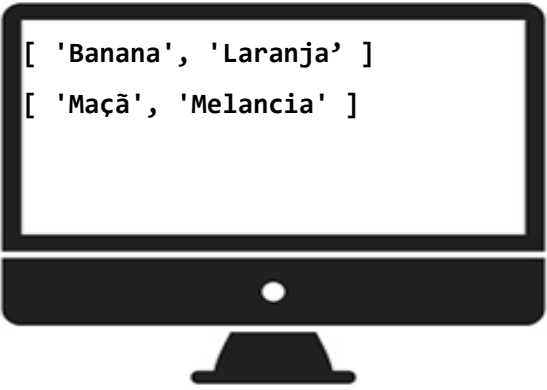
```
// Remove 1 elemento, a  
partir da posição 3
```





```
const frutas = [ 0'Banana', 1'Maçã', 2'Melancia', 3'Laranja'];  
let removidos = frutas.splice(1, 2);  
console.log(frutas, removidos);
```

Remove 2 elementos,
a partir da posição 1.



```
// Usando índices negativos  
const frutas = [ -4'Banana', -3'Maçã', -2'Melancia', -1'Laranja'];  
let removidos = frutas.splice(-3, 1);  
console.log(frutas, removidos);
```

Remove 1 elemento,
a partir da posição -3.

Ao usar índices negativos, o
método inicia em -1 e decrece

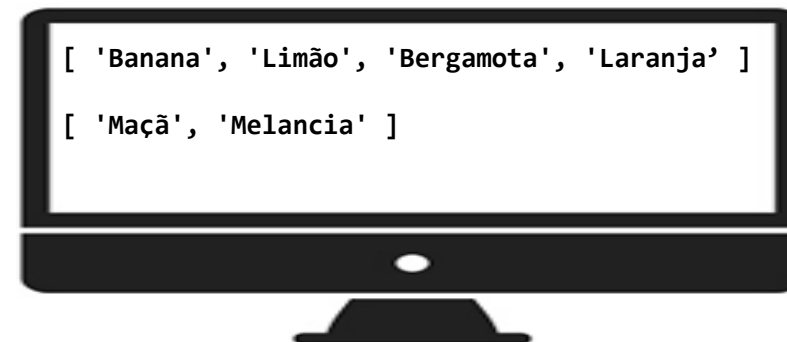


O uso de índices negativos no método splice() é uma forma
conveniente de acessar elementos começando do final do array.



```
// Removendo elementos e adicionando novos elementos
const frutas = [
  0      1      2      3
  'Banana', 'Maçã', 'Melancia', 'Laranja'
];
let removidos = frutas.splice(1, 2, 'Limão', 'Bergamota');
console.log(frutas, removidos);
```

Remove 2 elementos, a partir da posição 1, e adiciona também a partir de 1, os elementos 'Limão' e 'Bergamota'.



Como foi visto, o método **splice()** é utilizado para modificar o conteúdo de um array removendo, substituindo ou adicionando novos elementos.

Lembrando a sintaxe: **array.splice(início, quantos, item1, item2, ...)**, onde:

- **início:** O índice onde as modificações começam >> 1 significa que a operação começa no segundo elemento do array, que é 'Maçã'.
- **quantos:** O número de elementos a serem removidos. 2 significa que dois elementos serão removidos a partir do índice 1, ou seja, 'Maçã' e 'Melancia'.
- **item1, item2, ...:** Os novos elementos a serem adicionados no lugar dos removidos. Aqui, 'Limão' e 'Bergamota' serão inseridos no lugar de 'Maçã' e 'Melancia'.



```
// Simulando o push (inserção no final da lista)
```

```
const frutas = [0'Banana', 1'Maçã', 2'Melancia', 3'Laranja'];
```

```
let removidos = frutas.splice(frutas.length, 0, 'Mamão');
```

```
console.log(frutas, removidos);
```

Remove 0 elementos, e insere o elemento 'Mamão' a partir da posição 4, que é dada pelo atributo length.



O array **removido** fica vazio, já que não ocorreram remoções.

```
// Simulando o unshift (inserção no início da lista)
```

```
const frutas = [0'Banana', 1'Maçã', 2'Melancia', 3'Laranja'];
```

```
let removidos = frutas.splice(0, 0, 'Mamão');
```

```
console.log(frutas, removidos);
```

Não remove nenhum elemento, e insere o elemento 'Mamão' a partir da posição 0.



O array **removido** fica vazio, já que não ocorreram remoções.



slice() x splice()

Característica	slice()	splice()
Modifica o array original?	Não.	Sim.
Retorno do método?	Um novo array com os elementos extraídos.	Os elementos removidos (se houver).
Finalidade	Copiar parte de um array.	Remover, adicionar ou substituir elementos no array.
Sintaxe	<code>array.slice(início, fim)</code>	<code>array.splice(índice, quantos, item1, item2, ...)</code>

Resumo:

- **slice()** é indicado quando queremos **copiar** parte de um array ou criar um novo array sem modificar o original.
- **splice()** é indicado quando é necessário **modificar o array original** — seja para adicionar, remover ou substituir elementos.



`splice()` x `pop()`, `push()`, `shift()`, `unshift()`

Embora `splice()` possa replicar as funcionalidades desses métodos, sua **grande vantagem** é que ele permite **inserir, remover ou substituir** elementos em **qualquer posição do array**.

Esses outros métodos não fazem diretamente.

Resumo

- Adote `pop()`, `push()`, `shift()`, `unshift()` quando você estiver lidando com operações no **início ou final** do array. **Esses métodos são simples e diretos para esse tipo de operação.**
- Utilize `splice()` quando precisar de mais **flexibilidade**, como remover, adicionar ou substituir elementos **em qualquer posição** do array.

Matrices

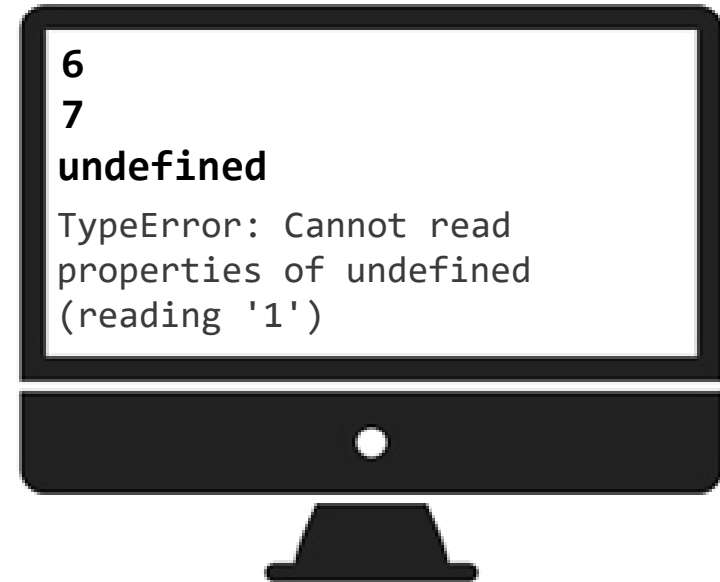




array multidimensional (matrizes)

Os arrays em **JavaScript** podem conter outros **arrays**, formando **arrays** multidimensionais (**arrays** de **arrays**)..

```
const matriz = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];  
  
console.log(matriz[1][2]);  
  
console.log(matriz[2][0]);  
  
console.log(matriz[2][3]);  
  
console.log(matriz[3][1]);
```





arrays aninhados e propriedades

Em **JavaScript**, ao acessarmos uma matriz multidimensional como `matriz[0][1]`, o que está acontecendo é :

```
const matriz = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];
```

`matriz[0]` acessa o primeiro **elemento**
da **matriz principal**, que é um array
(neste caso, a primeira "linha").

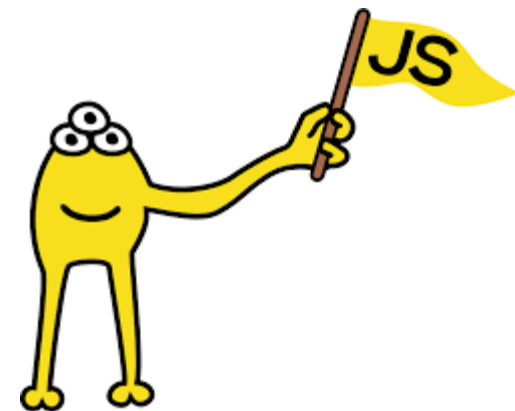
```
console.log(matriz[0][1]);
```

`matriz[0][1]` então acessa o **elemento no índice 1** do array que está em `matriz[0]`, ou seja, o segundo elemento da "linha" 0.

É possível pensar o conceito em termos de **propriedades**:

- o elemento `matriz[0]` é como se fosse um **objeto**; e
- o segundo índice, `[1]`, acessa uma **propriedade** desse objeto.

Embora tecnicamente estejamos trabalhando com arrays, o conceito de **propriedades** ainda se aplica, porque JavaScript trata arrays como objetos especiais onde os índices são **propriedades numeradas**.





Relevância de compreensão do comportamento

1) Estrutura de Dados Baseada em Objetos:

Arrays em **JavaScript** são, na verdade, **objetos** onde os índices são as propriedades numéricas.

Então, ao acessar `matriz[0]`, estamos acessando a propriedade 0 do objeto `matriz`, que contém outro objeto (um *array*).

```
const matriz = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];
```

Isso é relevante, pois explica por que acessar `matriz[3][1]` pode resultar em um erro se `matriz[3]` for **undefined**.

Nessa situação estamos tentando acessar uma **propriedade de undefined**, o que leva a um **TypeError**, porque **undefined** não pode ter propriedades.

```
console.log(matriz[3][1]);
```

TypeError: Cannot read properties of undefined (reading '1')



Relevância de compreensão do comportamento

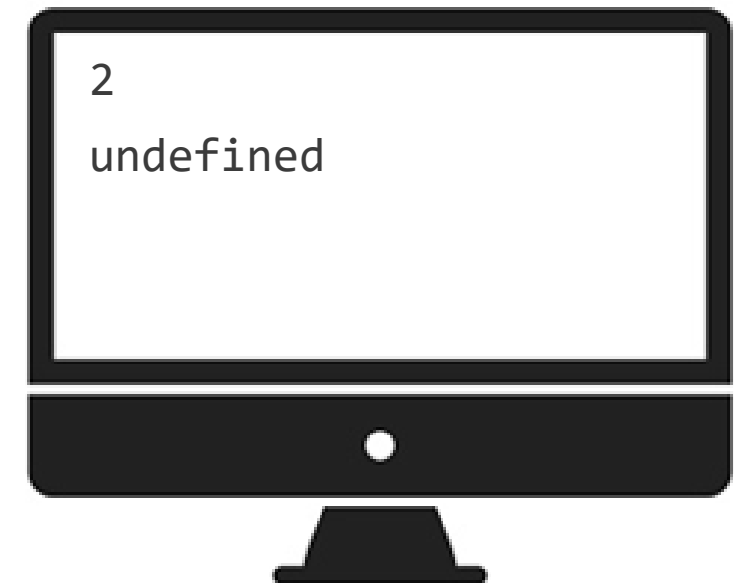
2) Matrizes Irregulares:

Como cada "linha" da matriz é um array por si só, elas podem ser **irregulares em termos de tamanho**.

Cada linha tem suas próprias propriedades (índices) que **podem variar de uma linha para outra**.

Isso significa que `matriz[0][1]` pode ser válido enquanto `matriz[2][1]` retorna **undefined**.

```
const matriz = [  
  [1, 2],    // Primeira linha tem 2 colunas  
  [3, 4, 5], // Segunda linha tem 3 colunas  
  [6]        // Terceira linha tem 1 coluna  
];  
  
console.log(matriz[0][1]);  
  
console.log(matriz[2][1]);
```





Relevância de compreensão do comportamento

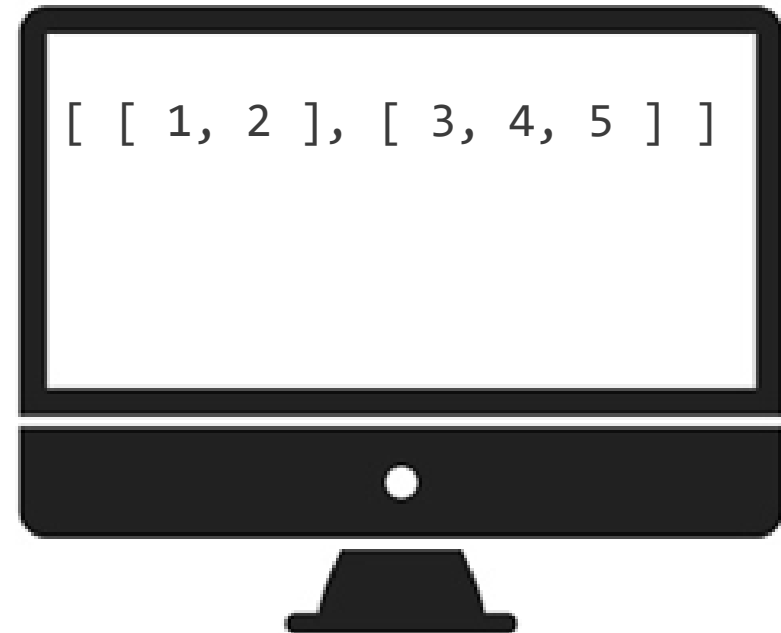
3) Manipulação direta de linhas.

Já que cada linha (array) é tratada como uma **propriedade** de um índice do array principal, é **possível manipular essas "linhas" diretamente** como seria feito com qualquer outro objeto ou array em **JavaScript**.

```
const matriz = [[1, 2], [3, 4]];

matriz[1].push(5);
// Adiciona um elemento à segunda linha

console.log(matriz);
```





Relevância de compreensão do comportamento

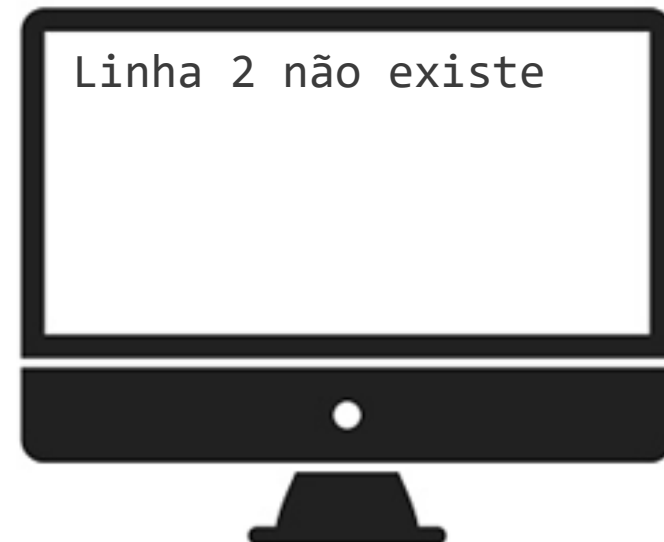
4) Verificação de existência.

Se uma linha não existir (`matriz[3]`), a tentativa de tentar acessar uma coluna dentro dessa linha **resultará em um erro**.

Isso reforça a ideia de que, em **JavaScript**, os arrays são tratados como objetos com propriedades, e a tentativa de acessar uma propriedade de **undefined** lançará um erro.

```
const matriz = [[1, 2], [3, 4]];

if (matriz[2]) {
  console.log(matriz[2][0]);
} else {
  console.log("Linha 2 não existe");
}
```



Valor x Referência

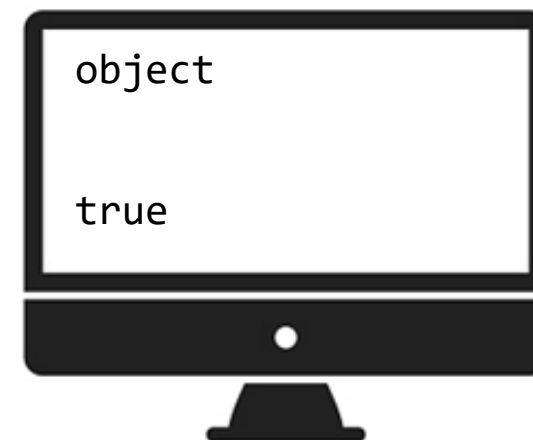




Descobrimos o tipo de dados: **typeof**

```
const alunos = ['Rafael', 'Maria', 'André', 'Ana', 'José', 'Anita'];  
console.log(typeof alunos);
```

typeof retorna que o **array** é um **object**,
como estudaremos mais a frente.



Mas, como saber se estamos trabalhando com um array?

```
console.log(alunos instanceof Array);
```

instanceof: Operador que verifica se o objeto que estamos
trabalhando **é uma instância (é do tipo) de uma classe específica**.

Retorna **true**, se verdadeiro, **false**, caso contrário.



Valor x Referência

Por ser um objeto, um array funciona **por referência**.

No JavaScript, uma variável pode armazenar dois tipos de valores: **primitivo** e **referência**.

Quando um valor é atribuído a uma variável, o motor JavaScript determina se o valor é **primitivo** ou de **referência**.

JavaScript tem **seis tipos primitivos**

undefined
null
boolean
number
string
symbol

O tamanho de valor primitivo é fixo, portanto, o JS armazena o valor primitivo na pilha (**stack**).

...e um tipo de referência:

object

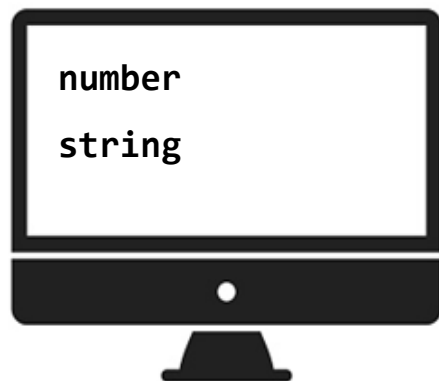
Já o tamanho de um valor de referência é dinâmico, o JS armazena o valor de referência no **heap**.

Se o valor for um tipo primitivo, ao acessar a variável, manipula-se o **valor real armazenado nessa variável**.

Em outras palavras, a variável que armazena um valor primitivo é **acessada por valor**.

Para se obter o **tipo de um primitivo**, se utiliza o operador **typeof**

```
let num = 13;  
console.log(typeof(num));  
let nome = 'tsi';  
console.log(typeof(nome));
```

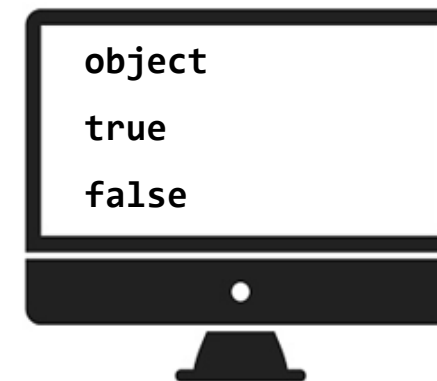


Ao manipular um objeto, **trabalha-se com a referência desse objeto**, e não o objeto real.

Isso significa que uma variável que armazena um objeto é **acessada por referência**.

Para saber se o **tipo é um valor de referência** (objeto), se utiliza o operador **instanceof**

```
let vetor = ['Stitch', 'Maria', 'Joana'];  
console.log(typeof(vetor));  
console.log(vetor instanceof Array);  
console.log(vetor instanceof String);
```





Ao atribuir uma variável que armazena um valor primitivo a outra variável, **este valor na verdade é criado e copiado na nova variável.**

```
let a = 13;
```

```
let b = a;
```

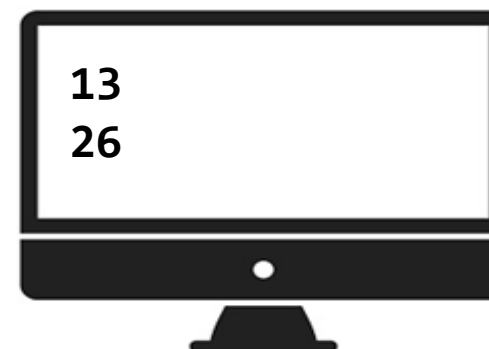
```
b = 26;
```

```
console.log(a);
```

```
console.log(b);
```



Copia o valor de a e armazena em b.





Ao atribuir uma variável que armazena um valor de referência a outra variável, **este valor também é copiado para a nova variável**.

A diferença é que os valores nas variáveis são os **endereços do objetos reais armazenados no *heap***.

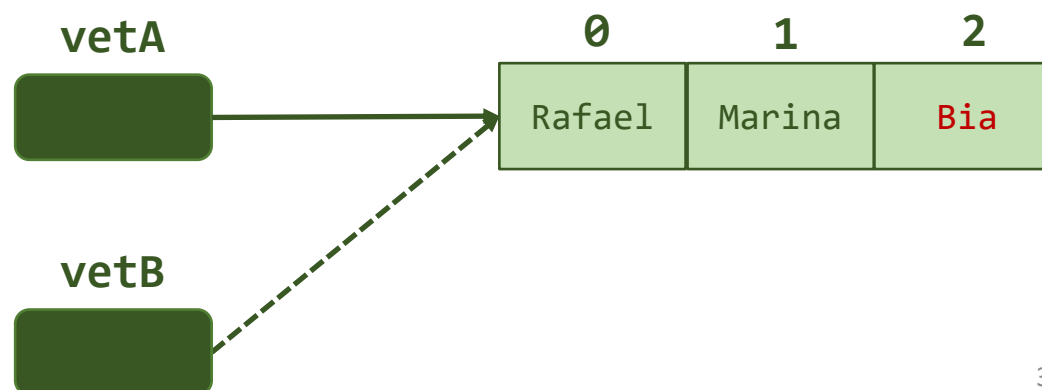
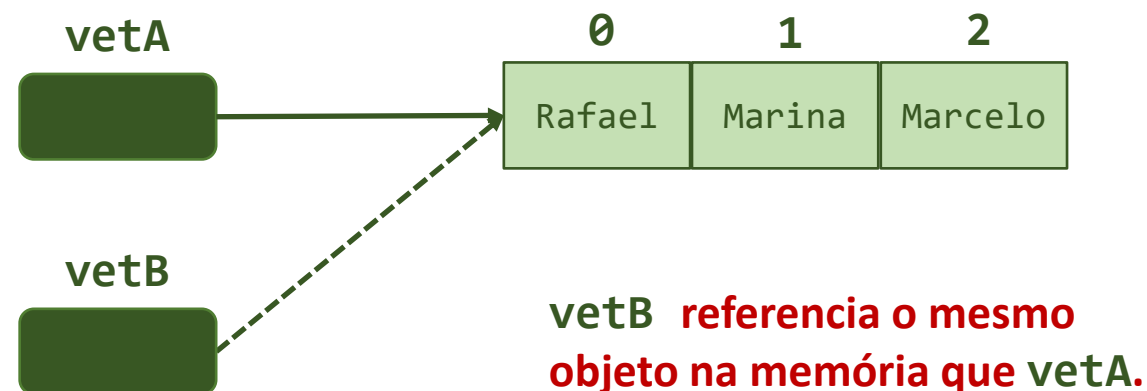
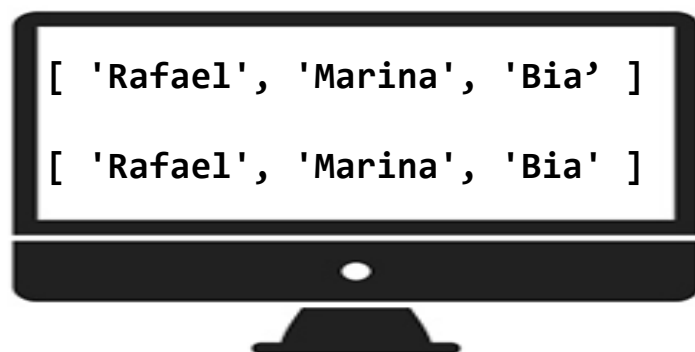
```
let vetA = ['Rafael', 'Marina', 'Marcelo'];
```

```
let vetB = vetA;
```

```
vetB[2] = 'Bia';
```

```
console.log(vetA);
```

```
console.log(vetB);
```



...spread





spread operator (...)

E caso seja necessário **copiar apenas os dados de um array para outro**, sem compartilhar a referência?

spread

Copia os dados do array original e “**espalha estes dados**” em um novo array.

Desta forma eles **não compartilham a mesma referência**. Estamos criando um novo array.


```
let vetOriginal = ['Rafael', 'Marina', 'Marcelo'];
```

```
let vetCopia = [...vetOriginal];
```

```
vetCopia.pop();
```

```
console.log(vetOriginal);
```

```
console.log(vetCopia);
```

A black computer monitor with a white screen. The screen displays two lines of JavaScript array literals. The first line is `['Rafael', 'Marina', 'Marcelo']` and the second line is `['Rafael', 'Marina']`.

```
[ 'Rafael', 'Marina', 'Marcelo' ]  
[ 'Rafael', 'Marina' ]
```

Concatenando arrays





Concatenando arrays método – concat()

```
const array1 = [1, 2, 3];
```

```
const array2 = [4, 5, 6];
```

```
let array3 = array1.concat(array2)
```

```
console.log(array3);
```

```
array3 = array1.concat([7, 8, 9]);
```

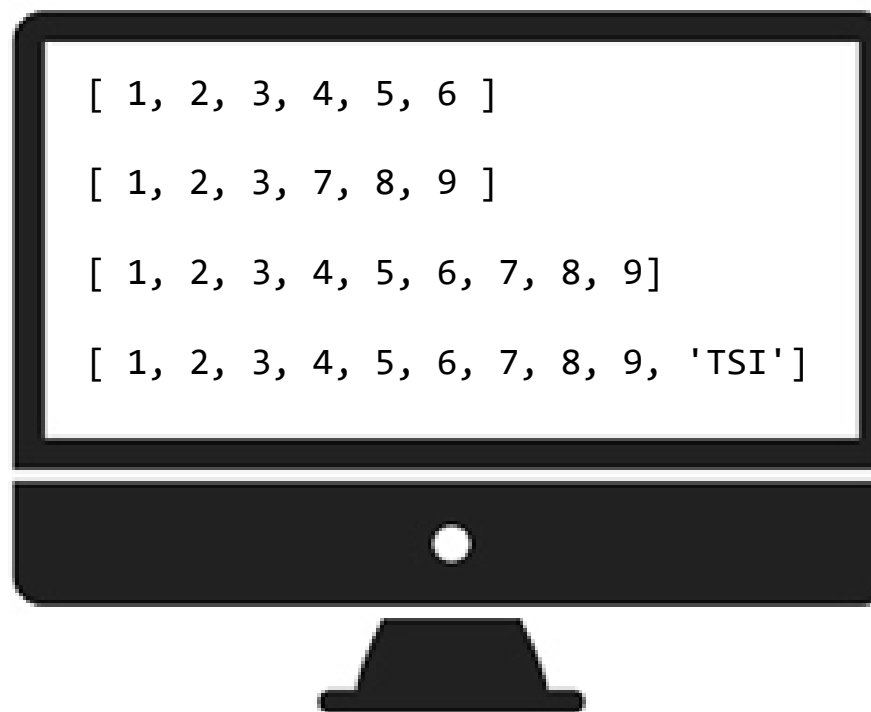
```
console.log(array3);
```

```
array3 = array1.concat(array2, [7, 8, 9]);
```

```
console.log(array3);
```

```
array3 = array1.concat(array2, [7, 8, 9], 'TSI');
```

```
console.log(array3);
```



concat() é utilizado para mesclar dois ou mais arrays. O método **não altera os arrays existentes**, em vez disso, **retorna um novo array**.



Concatenando arrays com **spread**

```
const array1 = [1, 2, 3];
```

```
const array2 = [4, 5, 6];
```

```
let array3 = [...array1, ...array2];
```

```
console.log(array3);
```

```
array3 = [...array1, [0, 0, 0]];
```

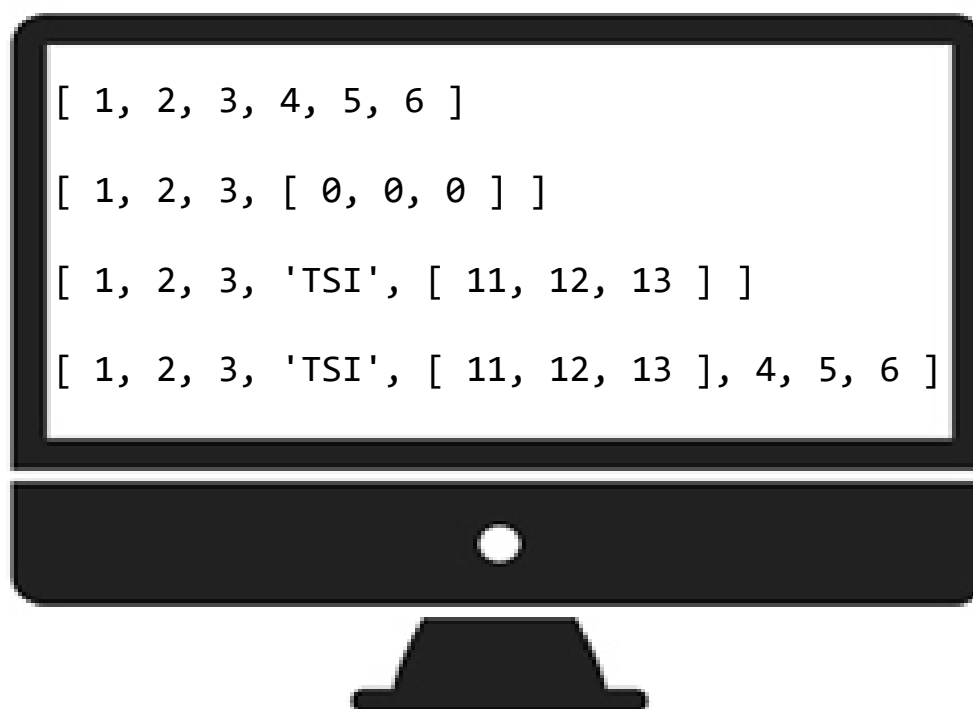
```
console.log(array3);
```

```
array3 = [...array1, 'TSI', [11, 12, 13]];
```

```
console.log(array3);
```

```
array3 = [...array1, 'TSI', [11, 12, 13], ...array2];
```

```
console.log(array3);
```



spread, espalha um array dentro do outro.

filter()





filter() método utilizado para filtrar arrays.

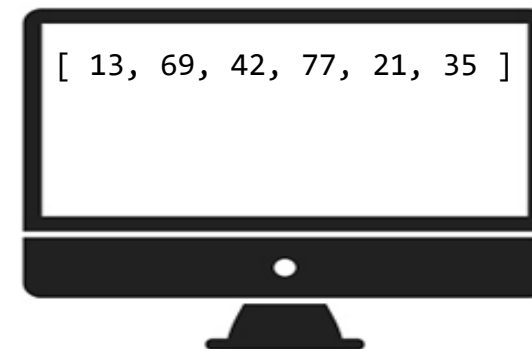
```
//           0  1  2  3  4  5  6  7  8  9  
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

Problema: retorne um novo array, com apenas os **valores maiores que 10**.

```
function callbackFilter(valor, indice, array) {  
  if(valor>10){  
    return true;  
  }else{  
    return false;  
  }  
}
```

Recebe 3 parâmetros de entrada:

- a. O elemento atual que será iterado;
- b. O índice do elemento
- c. O array completo



```
let numFiltrados = num.filter( callbackFilter );  
console.log(numFiltrados);
```

Método **filter()** cria um novo array, baseado em um filtro.

Quem filtra? O método **filter** recebe como parâmetro **uma função que itera por cada um dos elementos do array**, permitindo a realização do filtro.

filter() gera um novo array, composto pelos elementos que **satisfazem uma determinada condição**.



Simplificando a callbackfilter()

```
//      0   1   2   3   4   5   6   7   8   9  
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
function callbackFilter(valor, indice, array) {  
  if(valor>10){  
    return true;  
  }else{  
    return false;  
  }  
}
```

```
let numFiltrados = num.filter( callbackFilter );  
console.log(numFiltrados);
```

4
É possível **mudar para**
sintaxe de arrow function

1 Não estamos usando **indice** e **valor** na lógica da **function**. Assim, é **possível removê-los**.

```
function callbackFilter(valor) {  
  return valor > 10;  
}
```

2 É possível aprimorar a
lógica da function
para simplifica-la.

```
numFiltrados = num.filter(function (valor) {  
  return valor > 10;  
});
```

3
Se essa função não é utilizada
em outro lugar, é **possível**
transformá-la em anônima.

```
numFiltrados = num.filter((valor)=>{  
  return valor > 10;  
});
```



Reduzindo ainda mais a callbackfilter()

```
//           0   1   2   3   4   5   6   7   8   9  
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
numFiltrados = num.filter((valor)=>{  
    return valor > 10;  
});
```

```
console.log(numFiltrados);
```



```
numFiltrados = num.filter(valor => {  
    return valor > 10;  
});
```



```
numFiltrados = num.filter(valor => valor > 10);
```

Função completa que checa todos os elementos do array, verificando quais são maiores que 10, e retornando o novo array.

```
// Código final de filtro com call-back completo  
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
numFiltrados = num.filter(valor => valor > 10);
```

```
console.log(numFiltrados);
```

5 Função só tem um argumento. Assim, é possível eliminar os parênteses da **arrow function**.

6

Função tem apenas uma linha, ou seja, só retorna um valor. É possível excluir as chaves e a palavra-chave **return** e o **';**.

map()





map()

Diferentemente de **filter()**, que retorna um array do mesmo tamanho ou menor, **map()** **retorna um array exatamente do mesmo tamanho, mas com os valores alterados (ou não).**

Problema: triplicar todos os valores de um array

```
//           0   1   2   3   4   5   6   7   8   9
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
let numEmTriplo = num.map(function(valor, indice, array){
  console.log(valor, indice);
});
```

map() funciona da mesma forma que **filter()**.

Só muda a chamada da função, que agora é um **map()**

Parâmetros de entrada:

- a. O elemento atual que será iterado;
- b. O índice do elemento
- c. O array completo



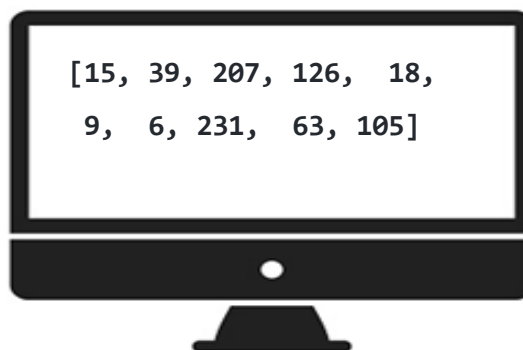
map() x filter()

filter() retorna *true* ou *false*, dependendo se elemento satisfaz uma condição, sendo adicionado ou não ao array final.

Por isso tem tamanho igual ou menor do que o array original.

```
//           0  1  2  3  4  5  6  7  8  9
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
let numEmTriplo = num.map(function(valor){
  return valor*3;
});
console.log(numEmTriplo);
```



map() retorna um novo valor para o elemento que está sendo processado.

Por isso que sempre terá o mesmo tamanho do array original.

```
//           0  1  2  3  4  5  6  7  8  9
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
let numEmTriplo = num.map(valor => valor * 3);
console.log(numEmTriplo);
```

Versão reduzida do **map()**

reduce()





reduce()

Método usado para **reduzir** um array a um único elemento.

Problema: somar todos os valores de um array

```
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
const total = num.reduce(function(accumulator, valor, indice, array){  
    console.log(accumulator);  
}, 0);
```

Além dos parâmetros que `map()` e `filter()` usam, existe um outro parâmetro no `reduce()`, o **acumulador**.

Opcional: Valor inicial do **acumulador**

Se não for enviado, ele assume o valor do primeiro item.

Recebe 3 parâmetros de entrada:

- a. O elemento atual que será iterado;
- b. O índice do elemento
- c. O array completo



reduce()

```
const num = [05, 113, 269, 342, 46, 53, 62, 777, 821, 935];
```

```
const total = num.reduce(function(accumulator, valor, indice, array){  
    accumulator += valor;  
    return accumulator;  
}, 0);  
console.log(total);
```





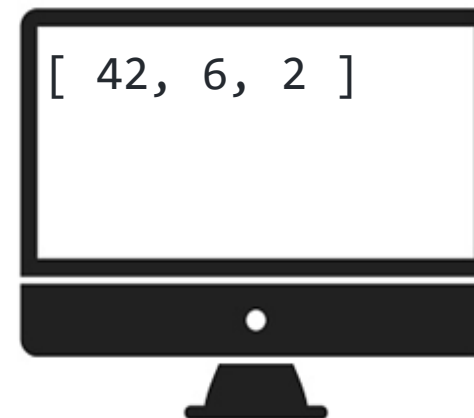
reduce() – outros usos

Problema: retornar um array com os números pares usando `reduce()`.

```
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

```
const arrayPares = num.reduce(function (acumulador, valor) {  
  if(valor%2==0) acumulador.push(valor);  
  return acumulador;  
}, []);
```

É possível configurar o valor inicial, por exemplo, com um array vazio. **E a partir daí inserir elementos baseados em algum condição.**



Obviamente o caminho mais sensato de resolver esse problema é usando o **filter()**.

Esta é apenas uma demonstração de outras formas de utilização do **reduce()**.

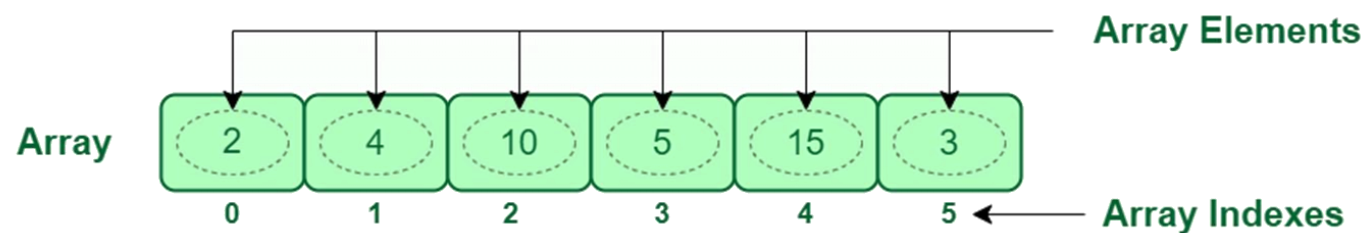


Desafio

Dado um array qualquer de números, retorne a soma do dobro de todos os números pares existentes no array.

```
const num = [5, 13, 69, 42, 6, 3, 2, 77, 21, 35];
```

Utilize as funções `filter()`, `map()` e `reduce()` para solucionar o problema.



arrays