Basic Data Structures / JAVASCRIPT

Usar um array para armazenar uma coleção de dados

Abaixo está um exemplo da implementação mais simples de uma estrutura de dados array. Isso é conhecido como *array unidimensional*, significando que tem apenas 1 nível de profundidade, ou que não possui nenhum outro array aninhado dentro de si. Note que possui *booleans*, *strings* e *numbers*, entre outros tipos de dados do JavaScript válidos:

```
let simpleArray = ['one', 2, 'three', true, false,
undefined, null];
console.log(simpleArray.length);
```

A chamada a `console.log` exibe `7`.

Todos os arrays possuem uma propriedade length, conforme mostrado acima, que pode ser muito facilmente acessado com a sintaxe `Array.length`. Uma implementação mais complexa de um array pode ser vista abaixo. Isso é conhecido como um *array multidimensional*, ou um array que contém outros arrays. Note que esse array também contém *objetos* JavaScript, os quais examinaremos bem de perto na próxima seção. Por agora, tudo que você precisa saber é que arrays também são capazes de armazenar objetos complexos.

```
let complexArray = [
  [
    {
      one: 1,
      two: 2
    },
    {
      three: 3,
      four: 4
    }
  ],
  [
    {
      a: "a",
      b: "b"
    },
```

```
    {
      c: "c",
      d: "d"
    }
  ]
];
```

EXERCICIO = Definimos uma variável chamada `yourArray`. Complete a instrução atribuindo um array de pelo menos 5 elementos de comprimento à variável `yourArray`. Seu array deve conter pelo menos uma *string*, um *número* e um *booleano*.

```
let yourArray = ["juju" , 2 , true , false, "flor"



] ;// Altere esta linha
```

Acessar o conteúdo de uma lista utilizando a notação de colchetes

A funcionalidade fundamental de qualquer estrutura de dados é, evidentemente, não só a capacidade de armazenar informação, como também a possibilidade de acessar esta informação quando necessário. Então, agora que aprendemos como criar um array, vamos começar a pensar em como podemos acessar as informações desse array.

Quando definimos um array simples como o que vemos abaixo, existem 3 itens nele:

```
let ourArray = ["a", "b", "c"];
```

Em um array, cada item do array possui um *índice* . Esse índice possui dois papéis: é a posição daquele item no array e como você o referencia. No entanto, é importante notar que arrays em JavaScript são *indexados a partir do zero*, o que significa que o primeiro elemento do array está, na verdade, na posição ***zero***, e não na primeira. Para recuperar um elemento de um array, nós podemos ao final de um array adicionar um índice encapsulado

com colchetes (por exemplo [0]), ou mais comumente, no final de uma variável que faz referência a um objeto array. Isso é conhecido como *notação de colchetes*. Por exemplo, se queremos recuperar o `a` de um array `ourArray` e atribuir a uma variável, nós podemos fazer isso com o código a seguir:

```
let ourVariable = ourArray[0];
```

Agora, `ourVariable` possui o valor de `a`.

Além de acessar o valor associado ao índice, você também pode *definir* um índice para um valor usando a mesma notação:

```
ourArray[1] = "not b anymore";
```

Usando a notação de colchetes, nós agora redefinimos o item no índice 1, alterando a string `b`, para `not b anymore`. Agora, `ourArray` é `["a", "not b anymore", "c"]`.

Exercicio = A fim de concluir esse desafio, defina a segunda posição (index `1`) do `myArray` como qualquer coisa que deseje, exceto a letra `b`.

```
let myArray = ["a", "b", "c", "d"];
// Altere apenas o código abaixo desta linha
myArray[1] = "juju e bela são trelosas";
// Altere apenas o código acima desta linha
console.log(myArray);
```

Adicionar itens em um array com push() e unshift()

O comprimento de um array, como os tipos de dados que pode conter, não são fixos. Arrays podem ser definidos com um comprimento de qualquer número de elementos e elementos podem ser adicionados e removidos com o passar do tempo. Em outras palavras, arrays são *mutáveis*. Nesse desafio, examinaremos dois métodos com os quais podemos modificar programaticamente um array: `Array.push()` e `Array.unshift()`.

Ambos os métodos recebem 1 ou mais elementos como parâmetros e adicionam esses elementos ao array no qual o método está sendo chamado; o método `push()` adiciona elementos ao final do array, e `unshift()` adiciona ao início. Considere o seguinte:

```
let twentyThree = 'XXIII';
let romanNumerals = ['XXI', 'XXII'];

romanNumerals.unshift('XIX', 'XX');
```

`romanNumerals` teria os valores `['XIX', 'XX', 'XXI', 'XXII']`.

```
romanNumerals.push(twentyThree);
```

`romanNumerals` teria os valores `['XIX', 'XX', 'XXI', 'XXII', 'XXIII']`. Note que nós também podemos passar variáveis, as quais nos permitem uma flexibilidade ainda maior na modificação dinâmica dos dados de nosso array.

Exercicio = Definimos uma função, `mixedNumbers`, na qual estamos passando o array como um argumento. Modifique a função usando `push()` e `unshift()` para adicionar `'I', 2, 'three'` no início do array e `7, 'VIII', 9` ao final para que o array retornado contenha a representação dos números de 1 a 9 em ordem.

```
function mixedNumbers(arr) {
  // Altere apenas o código abaixo desta linha
(arr).unshift ('I' , 2 , 'three');
(arr).push (7 , 'VIII', 9);
  // Altere apenas o código acima desta linha
  return arr;
}
```

```
console.log(mixedNumbers(['IV', 5, 'six']));
```

Remover itens de um array com pop() e shift()

Tanto `push()` e `unshift()` possuem métodos correspondentes que são quase opostos funcionais: `pop()` e `shift()`. Como você já pode ter adivinhado, em vez de adicionar, `pop()` *remove* um elemento do fim de um array, enquanto `shift()` remove um elemento do início. A diferença chave entre `pop()` e `shift()` e seus primos `push()` e `unshift()`, é que nenhum dos métodos recebe parâmetros. Cada um deles só permite que seja modificado um elemento por vez em um array.

Vamos dar uma olhada:

```
let greetings = ['whats up?', 'hello', 'see ya!'];

greetings.pop();
```

`greetings` teria o valor `['whats up?', 'hello']`.

```
greetings.shift();
```

`greetings` teria o valor `['hello']`.

Nós também podemos retornar o valor do elemento removido com qualquer método dessa forma:

```
let popped = greetings.pop();
```

`greetings` teria o valor `[]` e `popped` teria o valor `hello`.

Exercicio = Nós definimos uma função, `popShift`, a qual recebe um array como argumento e retorna um novo array. Modifique a função, usando `pop()` e `shift()`, para remover o primeiro e o último elemento do array passado como argumento, e atribua os valores removidos para suas variáveis correspondentes, para que o array retornado contenha seus valores.

```
function popShift(arr) {
  let popped = arr.pop();
  let shifted = arr.shift();
  return [shifted, popped];
}
```

```
console.log(popShift(['challenge', 'is', 'not',
'complete']));
```

Remover itens usando splice()

Pois bem. Aprendemos como remover elementos do início e do fim de arrays usando `shift()` e `pop()`, mas e se quisermos remover um elemento de algum lugar do meio? Ou remover mais de um elemento de uma vez? Bem, é aí que `splice()` pode ser útil. `splice()` nos permite fazer isso: **remover qualquer número de elementos consecutivos** de qualquer lugar no array.

`splice` pode receber 3 parâmetros, mas por agora, nós focaremos apenas nos 2 primeiros. Os dois primeiros parâmetros de `splice()` são inteiros que representam índices, ou posições, dos itens no array para o qual o método `splice()` está sendo chamado. Lembre-se: arrays são *indexados a zero*. Então, para indicar o primeiro elemento do array, usaríamos `0`. O primeiro parâmetro de `splice()` representa o índice no array do qual começar a remover elementos, enquanto o segundo parâmetro indica o número de elementos a serem removidos. Por exemplo:

```
let array = ['today', 'was', 'not', 'so', 'great'];

array.splice(2, 2);
```

Aqui, nós removemos 2 elementos, começando com o terceiro elemento (no índice 2). `array` teria o valor `['today', 'was', 'great']`.

`splice()` não apenas modifica o array do qual está sendo chamado, mas também retorna um novo array contendo os valores dos elementos removidos:

```
let array = ['I', 'am', 'feeling', 'really', 'happy'];

let newArray = array.splice(3, 2);
```

`newArray` tem o valor `['really', 'happy']`.

Exercicio = Iniciamos um array `arr`. Use `splice()` para remover elementos do `arr`, para que apenas contenha elementos que somam ao valor de `10`.

```
const arr = [2, 4, 5, 1, 7, 5, 2, 1];
// Altere apenas o código abaixo desta linha
arr.splice(1,4);
// Altere apenas o código acima desta linha
console.log(arr);
```

Adicionar itens usando splice()

Você se lembra de quando mencionamos no último desafio que `splice()` pode receber até três parâmetros? Bem, você pode usar o terceiro parâmetro, composto por um ou mais elementos, para adicionar algo ao array. Isso pode ser incrivelmente útil para mudar rapidamente de um elemento, ou um conjunto de elementos, para outro.

```
const numbers = [10, 11, 12, 12, 15];
const startIndex = 3;
const amountToDelete = 1;

numbers.splice(startIndex, amountToDelete, 13, 14);
```

```
console.log(numbers);
```

A segunda ocorrência de `12` é removida, e adicionamos `13` e `14` no mesmo índice. O array `numbers` agora seria `[ 10, 11, 12, 13, 14, 15 ]`.

Aqui, começamos com um array de números. Em seguida, passamos o seguinte para `splice()`: o índice no qual começar a deletar os elementos (3), o número de elementos a serem deletados (1) e os argumentos restantes (13, 14) serão inseridos com início no mesmo índice. Note que pode haver um número qualquer de elementos (separado por vírgulas) seguindo `amountToDelete`, cada um dos quais é inserido.

Exercicio = Definimos uma função, `htmlColorNames`, a qual recebe um array de cores HTML como argumento. Modifique a função usando `splice()` para remover os dois primeiros elementos do array e adicionar `'DarkSalmon'` e `'BlanchedAlmond'` em seus respectivos lugares.

```
function htmlColorNames(arr) {
  // Altere apenas o código abaixo desta linha
const startIndex = 0;
const amountToDelete = 2;
arr.splice(startIndex, amountToDelete, 'DarkSalmon',
'BlanchedAlmond');
  // Altere apenas o código acima desta linha
  return arr;
}
```

```
console.log(htmlColorNames(['DarkGoldenRod', 'WhiteSmoke',
'LavenderBlush', 'PaleTurquoise', 'FireBrick']));
```

Copiar itens de um array usando slice()

O próximo método que abordaremos é `slice()`. Em vez de modificar um array, `slice()` copia ou *extrai* um determinado número de elementos para um novo array, deixando o array em que o método é chamado inalterado. `slice()` recebe apenas 2 parâmetros — o primeiro é o índice aonde começar a extração e o segundo é o índice no qual parar a extração (a extração ocorrerá até esse índice mas não o incluirá). Considere isto:

```
let weatherConditions = ['rain', 'snow', 'sleet', 'hail',
'clear'];

let todaysWeather = weatherConditions.slice(1, 3);
```

`todaysWeather` teria o valor `['snow', 'sleet']`, enquanto `weatherConditions` ainda teria `['rain', 'snow', 'sleet', 'hail', 'clear']`.

Assim, criamos um novo array extraindo elementos de um array existente.

Exercicio = Definimos uma função, `forecast`, que recebe um array como argumento. Modifique a função usando `slice()` para extrair a informação do array passado como argumento e retorne um novo array contendo os elementos strings `warm` e `sunny`.

```
function forecast(arr) {
  // Altere apenas o código abaixo desta linha
   arr = arr.slice(2 , 4);
  return arr;
}


// Altere apenas o código acima desta linha
console.log(forecast(['cold', 'rainy', 'warm', 'sunny',
'cool', 'thunderstorms']));
```

Copiar um array com o operador spread

Enquanto `slice()` nos permite sermos seletivos sobre quais elementos de um array copiar, entre várias outras tarefas úteis, o novo operador *spread* do ES6 nos permite facilmente copiar *todos* os elementos de um array, em ordem, com uma sintaxe simples e altamente legível. A sintaxe de spread é simplesmente essa: `...`

Na prática, podemos usar o operador "spread" para copiar um array assim:

```
let thisArray = [true, true, undefined, false, null];
let thatArray = [...thisArray];
```

`thatArray` é igual a `[true, true, undefined, false, null]`. `thisArray` permanece inalterado e `thatArray` contém os mesmos elementos que `thisArray`.

Exercicio = Definimos uma função, `copyMachine` que recebe `arr` (um array) e `num` (um número) como argumentos. A função deve retornar um novo array composto de `num` cópias de `arr`. Fizemos a maior parte do trabalho para você, mas isso ainda não está certo. Modifique a função usando a sintaxe de spread para que ela funcione corretamente (dica: outro método já mencionado pode ser útil aqui!).

```
function copyMachine(arr, num) {
  let newArr = [];
  while (num >= 1) {
    // Altere apenas o código abaixo desta linha
 newArr.push([...arr]);
    // Altere apenas o código acima desta linha
    num--;
  }
  return newArr;
}
```

```
console.log(copyMachine([true, false, true], 2));
```

Combinar arrays com o operador spread

Outra grande vantagem do operador *spread* é a capacidade de combinar arrays, ou de inserir todos os elementos de um array em outro, em qualquer índice. Com sintaxe mais tradicional, podemos concatenar arrays, mas isso só nos permite combinar arrays no final de um e no início de outro. A sintaxe do spread torna a seguinte operação extremamente simples:

```
let thisArray = ['sage', 'rosemary', 'parsley', 'thyme'];

let thatArray = ['basil', 'cilantro', ...thisArray,
'coriander'];
```

`thatArray` teria o valor `['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander']`.

Usando a sintaxe de spread, acabamos de conseguir uma operação que teria sido mais complexa e mais extensa se tivéssemos utilizado métodos tradicionais.

Exercicio =Definimos uma função `spreadOut` que retorna a variável `sentence`. Modifique a função usando o operador *spread* para que ele retorne o array `['learning', 'to', 'code', 'is', 'fun']`.

```
function spreadOut() {
  let fragment = ['to', 'code'];
  let sentence = ['learning' , ...fragment, 'is', 'fun']; //
Altere esta linha
  return sentence;
```

```
}


console.log(spreadOut());
```

Verificar a presença de um elemento com indexOf()

Já que arrays podem ser alterados, ou *mutados*, a qualquer momento, não há garantia de onde um dado estará em um determinado array, ou se esse elemento sequer existe. Felizmente, o JavaScript nos fornece outro método integrado, `indexOf()`, que nos permite rapidamente e facilmente checar pela presença de um elemento em um array. `indexOf()` recebe um elemento como parâmetro, e quando chamado, retorna a posição, ou índice, daquele elemento, ou `-1` se o elemento não existe no array.

Por exemplo:

```
let fruits = ['apples', 'pears', 'oranges', 'peaches',
'pears'];

fruits.indexOf('dates');
fruits.indexOf('oranges');
fruits.indexOf('pears');
```

`indexOf('dates')` retorna `-1`, `indexOf('oranges')` retorna `2` e `indexOf('pears')` retorna `1` (o primeiro índice no qual cada elemento existe).

`Exercicio =` `indexOf()` pode ser incrivelmente útil para verificar rapidamente a presença de um elemento em um array. Definimos uma função, `quickCheck`, que recebe um array e um elemento como argumentos. Modifique a função usando `indexOf()` para que retorne `true` se o elemento passado existe no array, e `false` caso não exista.

```
function quickCheck(arr, elem) {
  // Altere apenas o código abaixo desta linha
```

```
 if (arr.indexOf(elem) >= 0) {
    return true;
  }
  return false;
}
  // Altere apenas o código acima desta linha



console.log(quickCheck(['squash', 'onions', 'shallots'],
'mushrooms'));
```

Iterar através de todos os itens de um array usando laços for

Às vezes quando trabalhando com arrays, é muito útil ser capaz de iterar sobre cada item para encontrar um ou mais elementos que podemos precisar, ou para manipular o array baseado em qual item de dados atende a determinados critérios. JavaScript oferece diversos métodos integrados que fazem iteração sobre arrays de formas ligeiramente diferentes para alcançar resultados diferentes (como `every()`, `forEach()`, `map()`, entre outros). Porém, a técnica mais flexível e que nos oferece a maior capacidade de controle é o laço `for` simples.

Considere o seguinte:

```
function greaterThanTen(arr) {
  let newArr = [];
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] > 10) {
      newArr.push(arr[i]);
    }
  }
  return newArr;
}
```

```
greaterThanTen([2, 12, 8, 14, 80, 0, 1]);
```

Usando o laço `for`, essa função itera o array, acessa cada elemento do array e submete-o a um teste simples que nós criamos. Dessa forma, nós determinamos de forma fácil e programática qual item é maior que `10`, e retornamos um novo array, `[12, 14, 80]`, contendo esses itens.

Exercicio = Definimos uma função, `filteredArray`, a qual recebe `arr`, um array aninhado, e `elem` como argumentos, e retornamos um novo array. `elem` representa um elemento que pode ou não estar presente em um ou mais dos arrays aninhados dentro de `arr`. Modifique a função, usando o laço `for`, para retornar uma versão filtrada do array recebido mode que qualquer array aninhado dentro de `arr` e contendo `elem` seja removido.

```
function filteredArray(arr, elem) {
  let newArr = [];
  // change code below this line


  for (let i = 0; i < arr.length; i++) {
    if (arr[i].indexOf(elem) == -1) {
      //Checks every parameter for the element and if is NOT
there continues the code
      newArr.push(arr[i]); //Inserts the element of the
array in the new filtered array
    }
  }


  // change code above this line
```

```
  return newArr;
}
// change code here to test different cases:
console.log(filteredArray([[3, 2, 3], [1, 6, 3], [3, 13, 26], [19, 3, 9]], 3));
```

OBS: O índice para iniciar a procura. Se o índice for maior ou igual ao tamanho do array, é retornado -1 e signfica que o item não será procurado. Se o `pontoInicial é fornecido` com um número negativo, é tomado como deslocamento da extremidade do array. Nota: se o `pontoInicial` fornecido é negativo, a procura no array acontece de frente para trás. Se o `pontoInicial` fornecido é 0, então o array inteiro será pesquisado. Padrão: 0 (pesquisa em todo array).

OBS : PARA ACESSAR PRIMEIRO SEMPRE VOU COLOCAR A LINHA E DEPOIS A COLUNA. SE EU QUISER VER A MATRIZ TODA DE UMA VEZ... É SÓ EU COLOCAR O NOME DA MATRIZ(OU ARRAY)

Criar arrays multidimensionais complexos

Excelente! Você acabou de aprender muito sobre arrays! Esta foi uma visão geral de nível bastante elevado, e há muito mais a aprender para trabalhar com arrays. Muitas dessas questões você verá em sessões posteriores. Mas antes de passarmos a examinar os *objetos*, vamos dar mais uma olhada e ver como os arrays podem se tornar um pouco mais complexos do que aquilo que vimos nos desafios anteriores.

Uma das características mais poderosas ao pensar em arrays como estruturas de dados é que arrays podem conter, ou mesmo ser completamente compostos por outros arrays. Vimos arrays que contêm arrays em desafios anteriores, mas que são bastante simples. No entanto, os arrays podem conter uma profundidade infinita de arrays que podem conter outros arrays, cada um com seus próprios níveis arbitrários de profundidade, e assim por diante. Desta forma, um array pode muito rapidamente se tornar uma estrutura de dados muito complexa, conhecida como *array multidimensional* ou array aninhado. Considere o seguinte exemplo:

```
let nestedArray = [
  ['deep'],
  [
    ['deeper'], ['deeper']
  ],
  [
    [
      ['deepest'], ['deepest']
    ],
    [
      [
        ['deepest-est?']
      ]
    ]
  ]
];
```

O array `deep` está aninhado com 2 níveis de profundidade. Os arrays `deeper` têm 3 níveis de profundidade. Os arrays `deepest` têm 4 níveis, e os `deepest-est?` têm 5 níveis.

Embora este exemplo possa parecer complicado, este nível de complexidade não é inédito, ou ainda fora do normal, em se tratando de grandes quantidades de dados. Entretanto, nós ainda podemos facilmente acessar os níveis mais profundos de um array complexo com a notação de colchetes:

```
console.log(nestedArray[2][1][0][0][0]);
```

Isso exibe no console a string `deepest-est?`. Agora que sabemos onde esse pedaço de dado está, nós podemos redefini-lo se precisarmos:

```
nestedArray[2][1][0][0][0] = 'deeper still';

console.log(nestedArray[2][1][0][0][0]);
```

Agora, ele mostra no console `deeper still`.

Exercicio = Definimos uma variável, `myNestedArray`, definida igual a um array. Modifique `myNestedArray`, usando qualquer combinação de *strings*,

numbers, e *booleans* para elementos, para que tenha 5 níveis de profundidade (lembre-se: o array mais extremo é de nível 1). Em algum lugar no terceiro nível, inclua a string `deep`, no quarto nível, inclua a string `deeper`, e no quinto nível, inclua a string `deepest`.

```
let myNestedArray = [
  // change code below this line
  ["unshift", false, 1, 2, 3, "complex", "nested"],
  ["loop", "shift", 6, 7, 1000, "method"],
  ["concat", false, true, "spread", "array", ["deep"]],
  ["mutate", 1327.98, "splice", "slice", "push",
[["deeper"]]],
  ["iterate", 1.3849, 7, "8.4876", "arbitrary", "depth",
[[["deepest"]]]]
  // change code above this line
];
```

Entendi, que a cada colchetes dentro do outro signifca um nivel de profundidade!

Adicionar pares de chave-valor a objetos JavaScript

Em suas formas mais básicas, objetos são apenas coleções de pares de *chave-valor*. Em outras palavras, eles são pedaços de dados (*valores*) mapeados para identificadores únicos chamados *propriedades* (*chaves*). Dê uma olhada no exemplo:

```
const tekkenCharacter = {
  player: 'Hwoarang',
  fightingStyle: 'Tae Kwon Doe',
  human: true
};
```

O código acima define um objeto de personagens do jogo de videogame Tekken chamado `tekkenCharacter`. Tem três propriedades, em que cada uma é mapeada para um valor específico. Se você quer adicionar uma propriedade adicional, como "origin", isso pode ser feito atribuindo `origin` ao objeto:

```
tekkenCharacter.origin = 'South Korea';
```

Isso usa a notação de ponto. Se você observar o objeto `tekkenCharacter`, ele agora incluirá a propriedade `origin`. Hwoarang também tinha cabelos cor de laranja, bem diferentes. Você pode adicionar essa propriedade com notação de colchetes fazendo:

```
tekkenCharacter['hair color'] = 'dyed orange';
```

A notação de colchete é necessária se sua propriedade tem um espaço nela ou se você deseja usar uma variável para nomear a propriedade. No caso acima, a propriedade está entre aspas para denotá-la como uma string e será adicionada exatamente como mostrada. Sem aspas, ela será avaliada como uma variável e o nome da propriedade será qualquer valor que a variável for. Aqui está um exemplo com uma variável:

```
const eyes = 'eye color';

tekkenCharacter[eyes] = 'brown';
```

Após adicionar todos os exemplos, o objeto ficará assim:

```
{
  player: 'Hwoarang',
  fightingStyle: 'Tae Kwon Doe',
  human: true,
  origin: 'South Korea',
  'hair color': 'dyed orange',
  'eye color': 'brown'
};
```

Exercicio = O objeto `foods` foi criado com três entradas. Usando a sintaxe de sua escolha, adicione mais três entradas a ele: `bananas` com um valor de `13`, `grapes` com um valor de `35` e `strawberries` com um valor de `27`.

```
let foods = {
  apples: 25,
  oranges: 32,
  plums: 28
};


// Altere apenas o código abaixo desta linha
foods['grapes'] = 35;
foods.bananas = 13;
foods['strawberries'] = 27 ;
// Altere apenas o código acima desta linha


console.log(foods);
```

OBS: Se eu colocar nos colchetes sem aspas, vai dar indefinido, pq vai ser como uma variável indefinida, ai eu precisaria colocar const primeiro pra depois colocar como variável !!!!

Modificar um objeto aninhado dentro de um objeto

Agora vamos dar uma olhada em um objeto ligeiramente mais complexo. Propriedades de objeto podem ser aninhadas para uma profundidade

arbitrária e os seus valores podem ser de qualquer tipo de dado suportado pelo JavaScript, incluindo arrays e até mesmo objetos. Considere o seguinte:

```
let nestedObject = {
  id: 28802695164,
  date: 'December 31, 2016',
  data: {
    totalUsers: 99,
    online: 80,
    onlineStatus: {
      active: 67,
      away: 13,
      busy: 8
    }
  }
};
```

`nestedObject` possui três propriedades: `id` (o valor é um número), `date` (o valor é uma string) e `data`(o valor é um objeto com sua estrutura aninhada). Enquanto estruturas podem se tornar rapidamente complexas, nós ainda podemos usar as mesmas notações para acessar as informações que precisamos. Para atribuir o valor `10` para a propriedade `busy` do objeto aninhado `onlineStatus`, nós usamos a notação de ponto para referenciar a propriedade:

```
nestedObject.data.onlineStatus.busy = 10;
```

Exercicio = Aqui nós definimos um objeto `userActivity`, o qual inclui outro objeto aninhado dentro dele. Defina o valor da chave `online` para `45`.

```
let userActivity = {
  id: 23894201352,
  date: 'January 1, 2017',
  data: {
    totalUsers: 51,
```

```
    online: 42
  }
};


// Altere apenas o código abaixo desta linha
userActivity.data.online = 45;
// Altere apenas o código acima desta linha


console.log(userActivity);
```

Acessar nomes de propriedades com notação de colchetes

No primeiro desafio, nós mencionamos o uso da notação de colchetes como uma forma de acessar valores das propriedades usando a avaliação de uma variável. Por exemplo, imagine que nosso objeto `foods` está sendo usado em um programa para a caixa-registradora de um supermercado. Nós temos algumas funções que definem `selectedFood` e nós queremos checar a presença da `selectedFood` em nosso objeto `foods`. Isso pode parecer assim:

```
let selectedFood = getCurrentFood(scannedItem);
let inventory = foods[selectedFood];
```

Esse código vai avaliar o valor armazenado na variável `selectedFood` e retorna o valor daquela chave no objeto `foods`, ou `undefined` se não estiver presente. Notação de colchetes é muito útil porque, às vezes, as propriedades de um objeto não são conhecidas antes da execução ou nós precisamos acessá-las de uma forma mais dinâmica.

Exercicio = Nós definimos uma função, `checkInventory`, a qual recebe um item escaneado como argumento. Retorne o valor atual da chave

`scannedItem` no objeto `foods`. Você pode assumir que apenas chaves válidas serão fornecidas como um argumento para `checkInventory`.

```
let foods = {
  apples: 25,
  oranges: 32,
  plums: 28,
  bananas: 13,
  grapes: 35,
  strawberries: 27
};


function checkInventory(scannedItem) {
  // Altere apenas o código abaixo desta linha
     return  foods[scannedItem];
  // Altere apenas o código acima desta linha
}


console.log(checkInventory("apples"));
```

Obs: quando é função, geralmente se usa return ....

Usar a palavra-chave delete para remover propriedades de objetos

Agora você sabe o que são objetos, seus recursos básicos e suas vantagens. Resumindo, ele são uma forma de armazenar chave-valor que provê uma forma flexível e intuitiva de estruturar dados, ***e***, eles fornecem um desempenho rápido para acessá-los. Ao longo do resto destes desafios, descreveremos diversas operações que você pode executar em objetos,

com a finalidade de torná-lo confortável ao usar essas estruturas de dados úteis em seus programas.

Nos desafios anteriores, nós adicionamos e modificamos os pares de chave-valor de objetos. Aqui veremos como podemos *remover* uma chave-valor de um obeto.

Vamos revisitar nosso objeto de exemplo `foods` uma última vez. Se quisermos remover a chave `apples`, podemos removê-lo usando a palavra-chave `delete` assim:

```
delete foods.apples;
```

Exercicio = Use a palavra-chave delete para remover as chaves `oranges`, `plums` e `strawberries` do objeto `foods`.

```
let foods = {
  apples: 25,
  oranges: 32,
  plums: 28,
  bananas: 13,
  grapes: 35,
  strawberries: 27
};


// Altere apenas o código abaixo desta linha
delete foods.oranges;
delete foods.plums;
delete foods.strawberries;
// Altere apenas o código acima desta linha
```

```
console.log(foods);
```

Verificar se um objeto tem uma propriedade

Agora podemos adicionar, modificar e remover as chaves dos objetos. Mas e se apenas quiséssemos saber se um objeto tem uma propriedade específica? O JavaScript nos fornece duas maneiras diferentes de fazer isso. Um usa o método `hasOwnProperty()` e o outro usa a palavra-chave `in`. Se tivermos um objeto `users` com uma propriedade de `Alan`, podemos verificar a sua presença de qualquer uma das seguintes maneiras:

```
users.hasOwnProperty('Alan');
'Alan' in users;
```

Ambos retornariam `true`.

Exercicio = Termine de escrever a função para que ela retorne `true` se o objeto passado a ela contiver todos os quatro nomes, `Alan`, `Jeff`, `Sarah` e `Ryan` e retorne `false` do contrário.

```
let users = {
  Alan: {
    age: 27,
    online: true
  },
  Jeff: {
    age: 32,
    online: true
  },
```

```
  Sarah: {
    age: 48,
    online: true
  },
  Ryan: {
    age: 19,
    online: true
  }
};


function isEveryoneHere(userObj) {
  if (
    userObj.hasOwnProperty("Alan") &&
    userObj.hasOwnProperty("Jeff") &&
    userObj.hasOwnProperty("Sarah") &&
    userObj.hasOwnProperty("Ryan")
  ) {
    return true;
  }
  return false;
}


console.log(isEveryoneHere(users));
```

Iterar através das chaves de um objeto com a declaração for...in

Às vezes, você pode precisar iterar através de todas as chaves dentro de um objeto. Isso requer uma sintaxe específica no JavaScript chamada de declaração *for...in*. Para nosso objeto `users`, isso pode se parecer como:

```
for (let user in users) {
  console.log(user);
}
```

Isso vai exibir no console `Alan`, `Jeff`, `Sarah` e `Ryan` - cada valor em sua própria linha.

Nessa declaração, definimos uma variável `user` e, como você pode ver, essa variável é redefinida durante cada iteração para cada chave do objeto conforme o comando se repete através do objeto, resultando em cada nome de usuário sendo exibido no console.

Observação: objetos não mantém uma ordem para as chaves armazenadas como arrays fazem. Portanto, a posição de uma chave em um objeto, ou a ordem relativa na qual ela aparece, é irrelevante quando referenciando ou acessando aquela chave.

Exercicio = Nós definimos uma função `countOnline` a qual aceita um argumento (um objeto users). Use a declaração *for...in* dentro dessa função para iterar o objeto users passado para a função, e retorne o número de users o qual possuam a propriedade `online` definida como `true`. Um exemplo de um objeto users o qual pode ser passado para `countOnline` é mostrado abaixo. Cada usuário terá a propriedade `online` com um valor `true` ou `false`.

```
{
  Alan: {
    online: false
  },
  Jeff: {
    online: true
  },
  Sarah: {
    online: false
```

```
  }
}
```

```
const users = {
  Alan: {
    online: false
  },
  Jeff: {
    online: true
  },
  Sarah: {
    online: false
  }
}


function countOnline(usersObj) {
  // Altere apenas o código abaixo desta linha
let result = 0;
  for (let user in usersObj) {
    if (usersObj[user].online === true) {
      result++;
    }
  }
  return result;
```

```
  // Altere apenas o código acima desta linha
}


console.log(countOnline(users));

const users = {
  Alan: {
    online: false
  },
  Jeff: {
    online: true
  },
  Sarah: {
    online: false
  }
}


function countOnline(usersObj) {
  // Altere apenas o código abaixo desta linha
let result = 0;
  for (let user in usersObj) {
    if (usersObj[user].online === true) {
      result++;
    }
  }
```

```
  return result;
  // Altere apenas o código acima desta linha
}


console.log(countOnline(users));
```

Gerar um array de todas as chaves de objeto com Object.keys()

Também podemos gerar um array o qual contém todas as chaves armazenadas em um objeto usando o método `Object.keys()` e passando um objeto como argumento. Isso retornará um array com strings representando cada propriedade do objeto. Novamente, não haverá uma ordem específica para as entradas no array.

Termine de escrever a função `getArrayOfUsers` para que retorne um array contendo todas as propriedades do objeto que receber como argumento.

```
let users = {
  Alan: {
    age: 27,
    online: false
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
```

```
    online: false
  },
  Ryan: {
    age: 19,
    online: true
  }
};


function getArrayOfUsers(obj) {
  // Altere apenas o código abaixo desta linha
return Object.keys(users);
  // Altere apenas o código acima desta linha
}


console.log(getArrayOfUsers(users));
```

Modificar o array armazenado em um objeto

Agora você já viu todas as operações básicas para os objetos JavaScript. Você pode adicionar, modificar e remover pares de chave-valor, verifique se a chave existe e itere sobre todas as chaves em um objeto. Conforme continuar aprendendo JavaScript, você verá aplicações de objetos ainda mais versáteis. Adicionalmente, as aulas de Estrutura de Dados localizadas na seção Preparação para Entrevista de Codificação do currículo também cobrem os objetos ES6 *Map* e *Set*, ambos são semelhantes a objetos comuns, mas fornecem alguns recursos adicionais. Agora que você aprendeu o básico de arrays e objetos, você está totalmente preparado para começar a resolver problemas mais complexos usando JavaScript!

Exercicio = Dê uma olhada no objeto que fornecemos no editor de código. O objeto `user` contém três chaves. A chave `data` contém 5 chaves, uma delas possui um array de `friends`. A partir disso, você pode ver como objetos são flexíveis assim como estruturas de dados. Nós começamos escrevendo a função `addFriend`. Termine de escrevê-la para que receba um objeto `user`, adicione o nome do argumento `friend` no array armazenado em `user.data.friends` e retorne esse array.

```
let user = {
  name: 'Kenneth',
  age: 28,
  data: {
    username: 'kennethCodesAllDay',
    joinDate: 'March 26, 2016',
    organization: 'freeCodeCamp',
    friends: [
      'Sam',
      'Kira',
      'Tomo'
    ],
    location: {
      city: 'San Francisco',
      state: 'CA',
      country: 'USA'
    }
  }
};
```

```
function addFriend(userObj, friend) {
  // Altere apenas o código abaixo desta linha
userObj.data.friends.push(friend);
  return userObj.data.friends;
  // Altere apenas o código acima desta linha
}


console.log(addFriend(user, 'Pete'));
```