Filter, Map e Reduce

Nós já entendemos que um array serve para armazenar uma sequência de dados e que podemos manipular esta sequência.

Entendemos também que podemos manipular isso, adicionando, removendo e realizando alterações com os métodos próprios do array.

E agora chegou a hora de entrar mais a fundo na nossa etapa de manipulação de dados utilizando funções mais avançadas que são o `filter`, `map` e `reduce`.

Em resumo os métodos:

- `filter`: Retorna um array com os valores filtrados.
- `map`: Retorna um array com os valores modificados.
- `reduce`: Retorna um array, número ou texto a partir do array.

`filter`

A primeira função do nosso canivete suíço de funções utilitárias de array é o filter.

Pelo o nome você já consegue desconfiar o que ele faz... Ele filtra 😃

Basicamente o `filter` permite executar uma função de *callback* para cada elemento do array.

Ou seja, ele vai executar um `for` que a cada passada o vai executar a sua função. Bem útil quando queremos analisar cada elemento e tomar uma ação de decisão se fica ou sai.

A sua função de *callback* deve retornar apenas `true` ou `false`. Assim quando for retornado verdadeiro a função `filter` saberá que este elemento pode entrar, senão deixa ele de fora.

O resultado da função `filter` sempre será um outro array.

Parece complicado!? Sim! Mas vem comigo...

```
var numeros = [10, 18, 1, 15];

var maiorQue10 = function(item) {
    return item > 10;
}

var novosNumeros = numeros.filter(maiorQue10);

console.log(novosNumeros);
// → [18, 15]
```

A função *callback* chamada `maiorQue10` recebe como parâmetro o elemento corrente desse array.

O `filter` pode calcular o que você bem entender, já que a função de *callback* é que determina as regras do jogo.

Vamos ver o exemplo da lista de nomes e deixar entrar apenas quem tem o nome iniciado com a letra Z.

```
var nomes = ["Bruno", "Zezinho", "Fulano", "Douglas"];

var convidados = nomes.filter(function(item){
    return item.charAt(0) == "Z";
});

console.log(convidados);
// → ["Zezinho"]
```

No exemplo acima a função *callback* compara a primeira letra, se for igual a "Z" está dentro.

Agora outro exemplo, filtro de pares e ímpares:

```
var numeros = [10, 18, 1, 15, 2, 12, 21, 33, 100];

var pares = function(item) {
    return !(item % 2);
}

var impares = function(item) {
    return item % 2;
}

var numerosPar = numeros.filter(pares);
var numerosImpar = numeros.filter(impares);

console.log(numerosPar);
// → [10, 18, 2, 12, 100]

console.log(numerosImpar);
// → [1, 15, 21, 33]
```

E este outro exemplo de locadora de vídeos, um filtro de filmes bons e ruins:

```
var filmes = [
    {titulo: 'Titanic', duracao: 195, nota: 7.5},
    {titulo: 'The Avengers', duracao: 203, nota: 9.5},
    {titulo: 'Bean', duracao: 90, nota: 6.5}
]

var notaCorte = 8;

var bons = function(item) {
```

```
    return item.nota >= notaCorte;
}

var ruins = function(item) {
    return item.nota < notaCorte;
}

var filmesBons = filmes.filter(bons);
var filmesRuins = filmes.filter(ruins);

console.log(filmesBons);
// → [{titulo: "The Avengers", duracao: 203, nota: 9.5}]

console.log(filmesRuins);
// → [{titulo: "Titanic", duracao: 195, nota: 7.5},
//    {titulo: "Bean", duracao: 90, nota: 6.5}]
```

E agora para o nosso famigerado exemplo de convidados, vamos filtrar os VIPs:

```
var convidados = [
    {nome: "Daniel", vip: true, idade: 21},
    {nome: "Gabriel", vip: true, idade: 54},
    {nome: "Felipe", vip: false, idade: 37}
]

var vips = convidados.filter(function(convidado) {
    return convidado.vip;
});

console.log(vips);
// → [{nome: "Daniel", vip: true, idade: 21},
//    {nome: "Gabriel", vip: true, idade: 54}]
```

O `filter` é uma das funções mais úteis do conjunto, mas ela pode ser um pouco lenta. Especialmente porque ela retorna um **novo** array, isso significa que arrays muito, muito, muito longos podem consumir memória excessivamente.

map

A função `map` traduz os valores de um array em outro, a função de "tradução" é uma função também *callback* que você irá programar. E por isso é tão flexível.

A sua função *callback* deve retornar o valor corrente modificado, ou seja, deve manipular e retornar.

Para exemplificar, vamos ver uma função `map` que multiplica por 2 cada valor do array:

```
var numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var multiplica = function(item) {
```

```
    return item * 2;
}

var dobrados = numeros.map(multiplica);

console.log(dobrados);
// → [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Bem tranquilo não!?

A função `map` pode realizar funções mais complexas, vejamos agora esta que traduz valores de temperatura de fahrenheit para célsius:

```
var fahrenheit = [0, 32, 45, 46, 47, 91, 93, 121];

var celcius = fahrenheit.map(function(item) {
    return Math.round((item - 32)*5/9);
});

console.log(celcius);
// → [-18, 0, 7, 8, 8, 33, 34, 49]
```

Este outro exemplo deixa todos os nomes da lista em maiúsculo:

```
var convidados = [{nome: "felipe", vip: false}, {nome: "danIEl", vip:
true}, {nome: "João", vip: true}];

var convidadosNormalizado = convidados.map(function(item) {
    return Object.assign(item, {nome: item.nome.toUpperCase()});
});

console.log(convidadosNormalizado);
// → [{nome: "FELIPE", vip: false},
//    {nome: "DANIEL", vip: true},
//    {nome: "JOÃO", vip: true}]
```

Na função acima o desafio ficou ainda mais divertido trabalhando com objetos.

A cada passada no array de convidados a função *callback* atribuí uma propriedade ao objeto existem. Explico...

O array que foi fornecido ao `map` continha as propriedades `nome` e `vip`. Certo?

Se fossemos simplesmente fazer isso:

```
var convidadosNormalizado = convidados.map(function(item) {
    return item.nome.toUpperCase();
```

```
});
```

O resultado seria isso: `["FELIPE", "DANIEL", "JOÃO"]`

Como você percebeu a propriedade `vip` desapareceu, ficando apenas um array com elementos em string.

Para resolver essa situação, nós podemos fazer o seguinte código:

```
var convidadosNormalizado = convidados.map(function(item) {
    return {nome: item.nome.toUpperCase(), vip: item.vip};
});
```

Agora o resultado é o esperado: `[{nome: "FELIPE", vip: false},{nome: "DANIEL", vip: true},{nome: "JOÃO", vip: true}]`

Só que você concorda que isso é feio? Porque...

Porque temos que lembrar de replicar todas as propriedades do objeto. Se amanhã ou depois outro programador inventar de adicionar a propriedade, por exemplo, `idade` o código vai remove-la (porque provavelmente você não replicar na função como foi feito no `vip`).

Aí que entra a função `Object.assign(destino, origem)`.

Essa função atribui a um objecto origem um outro objeto destino. Que no nosso caso é: `Object.assign(item, {nome: ...})`.

A cada passada do `map` a variável `item` assume o objeto atual e atribuí um objeto (que é fabricado na hora com o `{}`). Como esse objeto tem a mesma propriedade do destino, a função `Object.assign` sobrescreve a propriedade e valor atual.

Dessa forma nós alteramos apenas a propriedade `nome` do objeto, deixando para a função `Object.assign` copiar as outras propriedades e valores do objeto.

`reduce`

A função `reduce` funciona de uma forma mais flexível, ela pode assumir o papel de um modificador de array ou então de um agregador (que faz somas, médias, etc).

Como nas funções de `filter` e `map` ele manipula cada valor do array, porem com uma diferença que ele "carrega" a cada interação uma variável acumulativa. Veja os parâmetros da documentação oficial:

```
array.reduce(callback[, valorInicial])
```

- `callback` - Função que é executada em cada valor no array, recebe quatro argumentos:
  - `acumulador` - O valor retornado na última invocação do *callback*, ou o argumento Valor Inicial, se fornecido.
  - `valorAtual` - O elemento atual que está sendo processado no array.
  - `indice` - O índice do elemento atual que está sendo processado no array.
  - `array` - O array ao qual a função `reduce` foi chamada.

- `valorInicial` - Opcional. Objeto a ser usado como o primeiro argumento da primeira chamada da função `callback`.

Apenas lendo as instruções da documentação oficial parece um pouco complicado, isso é porque a função `reduce` é bem versátil. Por isso vamos desmistificar com vários exemplos seu funcionamento.

Vejamos o nosso primeiro exemplo de somatória de array:

```
var valores = [1.5, 2, 4, 10];

var somatoria = valores.reduce(function(total, item) {
  return total + item;
}, 0);

console.log(somatoria);
// → 17.5
```

No exemplo acima de somatória temos:

- `[1.5, 2, 4, 10]` - Como valores do array.
- `total` - Como variável acumulativa da funcão de *callback*.
- `item` - Como variável que armazena o valor atual.

Executando a função `reduce` acima passo a passo, teríamos algo desse tipo na sua execução:

- Passo 1 - `total = 0`; `item = 1.5`
- Passo 2 - `total = 1.5`; `item = 2`
- Passo 3 - `total = 3.5`; `item = 4`
- Passo 4 - `total = 7.5`; `item = 10`

O resultado final que consta na variável `somatoria` é de `17.5`.

A cada passo o `reduce` interage com o array e constrói a variável `total`.

No próximo exemplo podemos ver o uso do `reduce` mais uma vez, agora para calcular a média:

```
var valores = [1.5, 2, 4, 10];

var media = valores.reduce((total, item, indice, array) => {
  total += item;

  if (indice === array.length - 1) {
    return total / array.length;
  }

  return total;
}, 0);

console.log(media);
// → 4.375
```

No exemplo acima de cálculo de média temos:

- `[1.5, 2, 4, 10]` - Como valores do array.
- `total` - Como variável acumulativa da funcão de *callback*.
- `item` - Como variável que armazena o valor atual.
- `indice` - Como variável de indexação do array passado ao `reduce`.

Como o cálculo de média é a somatória dos números dividido pela quantidade, aproveitamos o valor de `indice` (que inicia com 0) para saber a quantidade atual de elementos (por isso `indice + 1`).

Executando a função `reduce` acima passo a passo, teríamos algo desse tipo na sua execução:

- Passo 1 - `total = 0`; `item = 1.5`; `indice = 0`
- Passo 2 - `total = Infinity`; `item = 2`; `indice = 1`
- Passo 3 - `total = 1.75`; `item = 4`; `indice = 2`
- Passo 4 - `total = 1.9166667`; `item = 10`; `indice = 3`

O resultado final que consta na variável `media` é de `2.9791666666666665`.

O bacana da função `reduce` é que a variável acumulativa (que estamos chamando de `total`) pode receber qualquer valor inicial. Nos exemplos estamos iniciando ela com o valor `0` porque estamos trabalhando com números e acumulando em um valor final.

Agora imagine que você pode iniciar com qualquer outro tipo de dado, por exemplo, arrays. Assim a variável acumulativa (`total`) passará a se comportar como tal.

Vamos começar a extrapolar com um exemplo que fizemos no `map` que foi os números dobrados.

```
var numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var dobrados = numeros.reduce(function(total, item) {
    return total.concat(item * 2);
}, []);

console.log(dobrados);
// → [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

O exemplo acima prova a versatilidade da função `reduce`, basicamente com a mesma lógica de funcionamento ela criou um array como retorno ao invés de um número (como foi nos exemplos anteriores).

Isso foi possível por conta do parâmetro opcional do `reduce` chamado `valorInicial`. Que nesse caso é iniciado com `[]`.

Ou seja, ao invés de iniciar o `total` com `0` (como fizemos nos exemplos anteriores de cálculo numéricos), iniciamos como um array.

Isso permitiu usar a função `concat` (função análoga ao `push`) para incluir os números dobrados.

Nessa linha de emular o funcionamento da função `map`, vamos simular o funcionamento do `filter`:

```
var numeros = [10, 18, 1, 15];

var novosNumeros = numeros.reduce(function(total, item) {
    if(item > 10) {
        total.push(item);
    }

    return total;
}, []);

console.log(novosNumeros);
// → [18, 15]
```

A função `reduce` é poderosa e por isso devemos tomar alguns cuidados:

1. Sempre possuir um `return`, isso irá garantir que o `reduce` consiga capturar o retorno da função *callback*.
2. Mesmo sendo opcional sempre procurar iniciar o valor de `valorInicial`.

`filter` + `map` + `reduce`

Se o `filter`, `map` e `reduce` isolados são poderosos, imagine juntar as forças!?

Isso só é possível porque o JavaScript permite criar funções usando programação funcional.

Basicamente esse paradigma de programação permite que você "emende" o retorno de uma função em outra. Ou seja, a saída, por exemplo, do `map` com a entrada do `reduce`.

Para explicar, vamos imaginar uma situação hipotética onde precisaremos primeiro gerar os números dobrados apenas de pares. O que for ímpar deverá ser descartado.

Já que você acompanhou até aqui, você já sabe que o `filter` vai isolar os pares e o `map` irá multiplicar. Fizemos isso isoladamente.

Mas como colocar tudo isso junto? É o que vamos ver no exemplo a seguir:

```
var numeros = [10, 18, 1, 15, 2, 12, 21, 33, 100];

var pares = function(item) {
    return !(item % 2);
}

var dobrados = function(item) {
    return item * 2;
}

var numerosDobrados = numeros.filter(pares).map(dobrados);

console.log(numerosDobrados);
// → [20, 36, 4, 24, 200]
```

Como você pode ver acima, usamos a saída do `filter` como entrada do `map`.

Isso permite "anexar" funções de tratamento de dados para gerar uma saída única. Por exemplo, anexar mais uma função `filter` para retornar apenas os números maiores que 10.

Veja:

```
var numeros = [10, 18, 1, 15, 2, 12, 21, 33, 100];

var pares = function(item) {
    return !(item % 2);
}

var dobrados = function(item) {
    return item * 2;
}

var maiores = function(item) {
    return item >= 10;
}

var numerosDobrados = numeros.filter(pares).map(dobrados).filter(maiores);

console.log(numerosDobrados);
// → [20, 36, 24, 200]
```

E mais, anexar ainda a função `reduce` para retornar a média dos números dobrados. Veja:

```
var numeros = [10, 18, 1, 15, 2, 12, 21, 33, 100];

var pares = function(item) {
    return !(item % 2);
}

var dobrados = function(item) {
    return item * 2;
}

var maiores = function(item) {
    return item >= 10;
}

var media = function(total, item, indice) {
  return (total + item) / (indice + 1);
}

var mediaDobrada =
numeros.filter(pares).map(dobrados).filter(maiores).reduce(media, 0);
```

```
console.log(mediaDobrada);
// → 54.333333333333336
```

Eu sei que isso não tem muito serventia agora, afinal pra que você quer a média dos números pares dobrados maiores que 10? Não sei...

Mas eu sei de uma coisa... Que essas três funções juntas irão te ajudar demais quando for lidar com dados vindos de API REST (retorno do banco de dados para seu aplicativo).

Criando nossas próprias funções de manipulação de dados

Vamos imaginar um retorno de dados de um cadastro de carros cores:

```
var carros = [{marca: 'Audi', cor: 'preto'},
              {marca: 'Audi', cor: 'branco'},
              {marca: 'Ferarri', cor: 'vermelho'},
              {marca: 'Ford', cor: 'branco'},
              {marca: 'Peugot', cor: 'branco'}];
```

E que agora você precise agrupar os carros por marca ou cor. Como fazer?

Uma das formas é utilizar o `reduce` para, a partir das propriedades do objeto, reduzir ao nível de chave e depois incluir os objetos. Falei complicado, vamos ver na prática:

```
var carros = [{marca: 'Audi', cor: 'preto'},
              {marca: 'Audi', cor: 'branco'},
              {marca: 'Ferarri', cor: 'vermelho'},
              {marca: 'Ford', cor: 'branco'},
              {marca: 'Peugot', cor: 'branco'}];

function groupBy(array, prop) {
  var value = array.reduce(function(total, item) {
    var key = item[prop];
    total[key] = (total[key] || []).concat(item);

    return total;
  }, {});

  return value;
};

var agrupados = groupBy(carros, 'marca');
console.log(agrupados);
// → {
//     "Audi": [
//       {marca: "Audi", cor: "preto"},
//       {marca: "Audi", cor: "branco"
//     ],
//     "Ferarri": [
```

```
//       {marca: "Ferarri", cor: "vermelho"}
//     ],
//     "Ford": [
//       {marca: "Ford", cor: "branco"}
//     ],
//     "Peugot": [
//       {marca: "Peugot", cor: "branco"}
//     ]
//   }
```

E eu sei que agrupar por marca parece estranho, agora vamos imaginar que recebemos isso do nosso banco de dados:

```
var produtos = [
  {id: 123, nome: 'Camiseta', cor: 'preto', tamanho: 'G', categoria:
'Vestuário'},
  {id: 456, nome: 'Tênis', cor: 'preto', tamanho: '41', categoria:
'Vestuário'},
  {id: 789, nome: 'Bola', cor: 'verde', tamanho: 'Único', categoria:
'Esporte'}
]
```

Não sei qual seu grau de fluência com banco de dados, API ou retorno de chamadas REST (sim, eu sei que pode ser bem baixo), mas isso é extremamente comum, um retorno que, as vezes precisa ser trabalhado para ser exibido em um componente de frontend.

Esses componentes podem ser listas, gráficos, etc.

Vejamos como usar uma função personalizada de manipulação, chamada `groupBy`, na sua versão 1.0:

```
var produtos = [
  {id: 123, nome: 'Camiseta', cor: 'preto', tamanho: 'G', categoria:
'Vestuário'},
  {id: 456, nome: 'Tênis', cor: 'preto', tamanho: '41', categoria:
'Vestuário'},
  {id: 789, nome: 'Bola', cor: 'verde', tamanho: 'Único', categoria:
'Esporte'}
]

function groupBy(array, prop) {
  var value = array.reduce(function(total, item) {
    var key = item[prop];
    total[key] = (total[key] || []).concat(item);

    return total;
  }, {});

  return value;
};
```

```
var agrupados = groupBy(produtos, 'categoria');
console.log(agrupados);
// → {
//     "Esporte": [
//       {id: 789, nome: "Bola", cor: "verde", tamanho: "Único", categoria:
"Esporte"}
//     ],
//     "Vestuário": [
//       {id: 123, nome: "Camiseta", cor: "preto", tamanho: "G", categoria:
"Vestuário"},
//       {id: 456, nome: "Tênis", cor: "branco", tamanho: "41", categoria:
"Vestuário"}
//     ]
//   }
```

Basicamente esta função agrupa em um array do tipo hash/map pela categoria que você desejar, no caso acima agrupamos por categoria.

Vejamos agora, a versão 2.0 da função `groupBy` utilizando Prototype juntamente com o `filter`, veja o código e a explicação sobre isso abaixo:

```
var produtos = [
  {id: 123, nome: 'Camiseta', cor: 'preto', tamanho: 'G', categoria:
'Vestuário'},
  {id: 125, nome: 'Shorts', cor: 'preto', tamanho: 'G', categoria:
'Vestuário'},
  {id: 456, nome: 'Tênis', cor: 'preto', tamanho: '41', categoria:
'Vestuário'},
  {id: 789, nome: 'Bola', cor: 'verde', tamanho: 'Único', categoria:
'Esporte'}
]

Array.prototype.groupBy = function(prop) {
    var value = this.reduce(function(total, item) {
    var key = item[prop];
      total[key] = (total[key] || []).concat(item);

      return total;
    }, {});

    return value;
}

var produtosFiltrados = produtos.filter(function(item) {
    return item.cor == 'preto';
}).filter(function(item) {
    return item.tamanho == 'G';
}).groupBy('categoria');

console.log(produtosFiltrados);
```

```
// → {
//     "Esporte": [
//       {id: 789, nome: "Bola", cor: "verde", tamanho: "Único", categoria:
"Esporte"}
//     ],
//     "Vestuário": [
//       {id: 123, nome: "Camiseta", cor: "preto", tamanho: "G", categoria:
"Vestuário"},
//       {id: 456, nome: "Tênis", cor: "branco", tamanho: "41", categoria:
"Vestuário"}
//     ]
//   }
```

No exemplo acima, nós utilizamos o Prototype para inserir uma nova função no Array. Que no caso é a função `groupBy`.

Na prática o Prototype funciona assim, imagine que no JavaScript tudo é objeto e todos descende de um único objeto pai (o Adão da parada toda).

O objeto Adão implementa uma propriedade especial chamada `prototype`, que a grosso modo, permite que você sobrescreva funções ou adicione novas a qualquer objeto filho de Adão, ou seja, todos.

Então praticamente qualquer coisa no JavaScript pode receber novas funções e propriedades mesmo que você não tenha escrito o código daquilo, que é o caso do objeto do tipo `Array`.

Para ficar com fru-frus seria mais "elegante" adicionar uma função no array do que ficar passando ele como parâmetro (que foi o caso na versão 1.0):

```
var produtos = [
  {id: 123, nome: 'Camiseta', cor: 'preto', tamanho: 'G', categoria:
'Vestuário'},
  {id: 125, nome: 'Shorts', cor: 'preto', tamanho: 'G', categoria:
'Vestuário'},
  {id: 456, nome: 'Tênis', cor: 'preto', tamanho: '41', categoria:
'Vestuário'},
  {id: 789, nome: 'Bola', cor: 'verde', tamanho: 'Único', categoria:
'Esporte'}
]

var agrupados = groupBy(produtos, 'categoria'); // maneira urca, versão 1.0
var agrupados = produtos.groupBy('categoria'); // maneira elegante, versão
2.0
```

Rodar tanto uma quanto outra irá funcionar de qualquer forma, mas olha a manha:

```
var produtos = [
  {id: 123, nome: 'Camiseta', cor: 'preto', tamanho: 'G', categoria:
'Vestuário'},
  {id: 125, nome: 'Shorts', cor: 'preto', tamanho: 'G', categoria:
```

```
'Vestuário'},
  {id: 456, nome: 'Tênis', cor: 'preto', tamanho: '41', categoria:
'Vestuário'},
  {id: 789, nome: 'Bola', cor: 'verde', tamanho: 'Único', categoria:
'Esporte'}
]

var produtosFiltrados = produtos.filter(function(item) {
  return item.cor == 'preto';
});

var produtosAgrupados = groupBy(produtosFiltrados, 'categoria'); // maneira
urca, versão 1.0
// ...
```

Eu não sei você, mas eu que tenho TOC não conseguiria ficar mais de 1 minuto olhando para o `groupBy` e imaginando o porque ele não pode ter a mesma forma de sintaxe que do `filter`, `map` ou `reduce`.

Por isso (eu sei que é uma baita de uma frescura), o melhor seria:

```
var agrupados = produtos.filter(function(item) {
  return item.cor == 'preto';
}).groupBy('categoria');
```

Ahhh respiro aliViádo 😃

Como você mesmo viu as possibilidades são infinitas!

Você pode pegar carona na programação funcional usando as práticas funções `filter`, `map`, `reduce` e até o `sort` para manipular resultados de uma forma que jamais conseguiria com outra linguagem que não seja dinâmica (C, por exemplo).

Ou... Criar as suas como fizemos com o `groupBy` 😃

Desafio

Adicione em nosso pequeno sistema de controle de convidados as funções de filtro para barrar as pessoas com menos de 18 anos e mais.

Que possa permitir agrupar entre gêneros para melhorar a organização de cobrança e ordenar por nome, já que no futuro evento o cliente comentou que "mulher paga meia" e que precisa estar ordenado para conferência mais rápida de nomes.

Resposta

```
Array.prototype.groupBy = function(prop) {
    var value = this.reduce(function(total, item) {
    var key = item[prop];
      total[key] = (total[key] || []).concat(item);

      return total;
    }, {});

    return value;
}

var convidados = [
    {nome: 'Felipe', idade: 37, genero: 'masculino'},
    {nome: 'Amanda', idade: 17, genero: 'feminino'},
    {nome: 'João', idade: 27, genero: 'masculino'},
    {nome: 'Daniel', idade: 21, genero: 'masculino'},
    {nome: 'Helena', idade: 21, genero: 'feminino'}
];

var convidadosFiltrados = convidados.sort(function(a, b) {
    return (a.nome > b.nome) ? 1 : -1; // 1 B precede A, -1 A precede B
}).filter(function(item) {
    return item.idade >= 18;
}).groupBy('genero');

console.log(convidadosFiltrados);
// → {
//     "feminino": [
//       {nome: "Helena", idade: 21, genero: "feminino"}
//     ],
//     "masculino": [
//       {nome: "Daniel", idade: 21, genero: "masculino"},
//       {nome: "Felipe", idade: 37, genero: "masculino"},
//       {nome: "João", idade: 27, genero: "masculino"}
//     ]
//   }
```