CAPÍTULO 3

MÉTODOS AUXILIARES PARA ARRAY

Antes de entrarmos de cabeça nas novas funcionalidades trazidas para o JavaScript, gostaria de fazer uma revisão de alguns métodos auxiliares para `Array` que foram inseridos inicialmente na versão ES5.1, e agora completamente incorporados no ES6.

Antes de serem integrados oficialmente, estes métodos eram implementados por bibliotecas externas, como o Lodash (https://lodash.com/) e Underscore.js (http://underscorejs.org). Entretanto, dada a sua popularidade na comunidade de desenvolvedores JavaScript, eles foram introduzidos oficialmente na especificação. Estes métodos nos auxiliarão no decorrer de todo livro, permitindo-nos escrever códigos mais legíveis e fáceis de dar manutenção.

Existe uma infinidade de métodos auxiliares para `Array` , porém daremos uma atenção especial aos mais utilizados:

- `forEach`
- `map`
- `filter`
- `find`
- `every`
- `some`
- `reduce`

Os métodos funcionam de forma semelhante, mas cada um possui propósitos bem distintos. A maior exceção à regra é o `reduce`, que possui a sintaxe e funcionamento um pouco diferente.

A principal meta destes métodos é substituir a clássica iteração utilizando o laço de repetição com `for`, tornar o código mais claro para quem está lendo, e deixar explícita qual a intenção da iteração. Ao final do capítulo, seremos capazes de lidar com `Array` de forma muito mais fácil.

Este conhecimento será útil não somente para seguir adiante com o livro, mas também contribuirá para sua vida profissional como desenvolvedor JavaScript, afinal, grande parte dos projetos hoje já adota estes métodos.

3.1 A MANEIRA TRADICIONAL DE ITERAR UM ARRAY

Uma das tarefas mais repetitivas na vida de um desenvolvedor é percorrer os registros contidos em uma lista. Fazemos isto por muitos motivos: encontrar um registro específico, fazer a soma dos registros, eliminar um registro, ordenar por ordem crescente, e por aí vai.

Para alcançar estes objetivos, estamos acostumados — na maior parte das vezes — a utilizar o laço de repetição `for`. Veja o exemplo a seguir:

```
var frutas = ['abacaxi', 'maça', 'uva'];
for(var i = 0; i < frutas.length; frutas++) {
  // corpo da iteração
}
```

O maior problema com esta abordagem é que é impossível saber qual o objetivo do corpo da iteração sem ver sua implementação. É

possível que seja uma iteração para buscar um elemento, listar todos, excluir, ordenar... Não tem como saber. Com os métodos auxiliares que veremos, não teremos mais este problema. Vamos ver como eles podem nos ajudar.

3.2 FOREACH

O `forEach` é o primeiro método que veremos. Ele é uma mão na roda para quando precisamos passar por todos os elementos de dentro de um `Array`. Considere o caso no qual temos de mostrar no console todos os itens de uma lista de nomes. Com ES5, utilizando o laço de repetição `for`, estamos acostumados a fazer algo assim:

```
var nomes = ['maria', 'josé', 'joão'];
for(var i = 0; i < nomes.length; i++) {
  console.log(nomes[i]);
}

// maria, josé, joão
```

Obtemos o resultado esperado, mas agora veja como podemos melhorar o código e ter o mesmo efeito. Para isso, invocamos a função `forEach` no `Array`, e passamos como parâmetro uma função de retorno que aceita um outro parâmetro. Neste exemplo, usaremos a função anônima `function(nome){...}`:

```
var nomes = ['maria', 'josé', 'joão'];
nomes.forEach(function(nome) {
  console.log(nome);
});

// maria, josé, joão
```

Repare no que aconteceu. Dentro do `forEach`, passamos uma função anônima de retorno, que costumamos chamar de função de `callback`. Ela é executada para cada elemento dentro da lista. A cada iteração, o valor da lista é atribuído à variável passada como

parâmetro no `callback` — no nosso caso, a variável `nome`.

Neste exemplo, somente emitimos no console seu valor. Mas nesta função de `callback`, podemos fazer qualquer coisa com o valor da variável, inclusive passá-la como parâmetro para outros métodos.

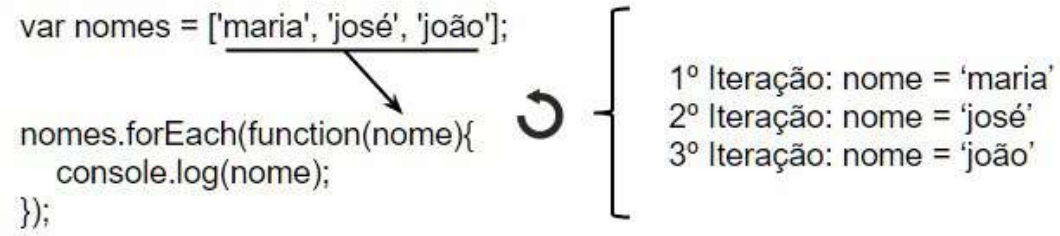

Figura 3.1: Iterando com o forEach

Entretanto, note que a função de `callback` não precisa necessariamente ser anônima. Podemos defini-la antes e atribuí-la a uma variável para passá-la como parâmetro ao `forEach`:

```
var nomes = ['maria', 'josé', 'joão'];

function imprimeNome(nome) {
  console.log(nome);
}

nomes.forEach(imprimeNome);
```

Em todos estes casos, a saída é exatamente a mesma:

```
maria
josé
joão
```

Bem legal, não é mesmo? Mas calma, nem tudo são flores. É preciso estar bastante atento ao fato de que os elementos processados pelo `forEach` são determinados antes da primeira invocação da função de `callback`. Isso significa que os elementos que forem adicionados depois da chamada do método não serão vistos.

O mesmo vale se os valores dos elementos existentes do `Array` forem alterados. O valor passado para o `callback` será o valor no momento em que o `forEach` visitá-lo. Avalie o código a seguir para entender o que isso quer dizer:

```
var canais = ["Globo", "Sbt", "Record"];
canais.forEach(function(canal) {
  canais.push("RedeTV"); // este item será ignorado
  console.log(canal);
})
```

Veja com atenção o que foi feito. Atribuímos a uma variável chamada `canais` uma lista que representa canais da televisão aberta brasileira. A seguir, invocamos o `forEach` e, dentro do `callback`, inserimos o canal `RedeTV` na nossa lista. Ao executar o código, podemos ver que a `RedeTV` nunca é exibida:

```
Globo
Sbt
Record
```

Isso acontece exatamente porque os elementos processados pelo `forEach` são determinados antes da primeira invocação da função de `callback`. Entretanto, isso não quer dizer que os valores não foram adicionados à lista. Ao adicionar um segundo `console.log` ao final do código para exibir a lista, notamos que a `RedeTV` foi adicionada várias vezes ao nosso `Array`. Uma cópia para cada iteração:

```
var canais = ["Globo", "Sbt", "Record"];
canais.forEach(function(canal) {
  canais.push("RedeTV"); // este item será ignorado
  console.log(canal);
})

console.log(canais);
// [ 'Globo', 'Sbt', 'Record', 'RedeTV', 'RedeTV', 'RedeTV' ]
```

3.3 MAP

O método `map` é muito útil quando precisamos não somente passar por todos os elementos de um `Array`, mas também modificá-los. Por exemplo, imagine que precisamos de um algoritmo para duplicar todos os valores de uma lista de números naturais. Sem pensar muito, faríamos algo assim:

```
var numeros = [1,2,3];
var dobro = [];

for(var i = 0; i < numeros.length; i++) {
  dobro.push(numeros[i] * 2);
}

console.log(numeros); // [1,2,3]
console.log(dobro); // [2,4,6]
```

Criamos um novo `Array` chamado `dobro` e usamos o seu método `push` para inserir o dobro de cada um dos valores recuperados por índice na iteração dos `numeros`. Podemos ter o mesmo comportamento ao usar o `map`:

```
var numeros = [1,2,3];
var dobro = numeros.map(function(numero) {
  return numero * 2;
});

console.log(numeros); // [1,2,3]
console.log(dobro); // [2,4,6]
```

O `map` executa a função de `callback` recebida por parâmetro para cada elemento iterado de `numeros` e constrói um novo `Array` com base nos retornos de cada uma das chamadas. Como o `map` nos devolve uma outra instância de `Array`, a lista original nunca é realmente modificada, o que mantém sua integridade.

E assim como no vimos no `forEach`, a função de `callback` não passa por elementos que foram modificados, alterados ou removidos depois da primeira execução da função de retorno.

3.4 FILTER

Como o próprio nome já pode induzir, este método é deve ser utilizado quando temos a necessidade de filtrar nossa lista de acordo com algum critério. Por exemplo, imagine que queremos filtrar de uma lista de alunos, todos os que são maiores de idade. Com o ES5, nós poderíamos fazer:

```
var alunos = [
  {nome:'joão', idade:15},
  {nome:'josé', idade:18},
  {nome:'maria', idade:20}
];

var alunosDeMaior = [];
for(var i = 0; i < alunos.length; i++) {
  if(alunos[i].idade >= 18) {
    alunosDeMaior.push(alunos[i]);
  }
}

console.log(alunosDeMaior);
// [{nome:'josé', idade:18}, {nome:'maria', idade:20}]
```

Com o método `filter`, temos o mesmo efeito de forma mais clara:

```
var alunos = [
  {nome:'joão', idade:15},
  {nome:'josé', idade:18},
  {nome:'maria', idade:20}
];

var alunosDeMaior = alunos.filter(function(aluno) {
  return aluno.idade >= 18;
});

console.log(alunosDeMaior);
// [{nome:'josé', idade:18}, {nome:'maria', idade:20}]
```

A função de `callback` recebe como parâmetro cada um dos alunos da lista em cada iteração — assim como aconteceu nas outras funções auxiliares que vimos — e o atribui na variável `aluno`. Dentro da função, utilizamos um critério de avaliação para devolver um valor booleano para o `filter`: `true` ou `false`. Se for

retornado verdadeiro, o valor é inserido no novo `Array` retornado; caso contrário, é simplesmente ignorado e não é incluído.

3.5 FIND

Esta função auxiliar é particularmente interessante quando o objetivo é encontrar um item específico dentro de um `Array`. Digamos, por exemplo, que de uma lista de alunos queremos somente o registro que contenha o nome "josé". O que faríamos tradicionalmente é algo nesse sentido:

```
var alunos = [
  {nome:'joão'},
  {nome:'josé'},
  {nome:'maria'}
];

var aluno;
for(var i = 0; i < alunos.length; i++) {
  if(alunos[i].nome === 'josé') {
    aluno = alunos[i];
    break; // evita percorrer o resto da lista
  }
}

console.log(aluno); // {"nome":"josé"}
```

Para cada elemento da lista, recuperamos a propriedade do elemento e o comparamos com o nome que estamos buscando. Se der igualdade, atribuímos o valor na variável `aluno` instanciada antes do loop e o encerramos. Com o `find`, é possível reescrever este código e obter o mesmo efeito, com a ressalva de que vamos pegar somente o primeiro item que satisfaz o critério de busca. Fica assim:

```
var alunos = [
  {nome:'joão'},
  {nome:'josé'},
  {nome:'maria'}
];
```

```
var aluno = alunos.find(function(aluno) {
  return aluno.nome === 'josé';
});

console.log(aluno); // {"nome":"josé"}
```

Caso na lista existissem dois alunos com o nome “josé”, somente o primeiro seria retornado. Para contornar este caso, precisaríamos usar um critério de busca mais específico.

3.6 EVERY

Esta é uma função auxiliar bem interessante. Ao contrário das outras que vimos até então, esta não retorna uma cópia do `Array`, mas sim um valor booleano.

A função `every` é pertinente para validar se todos os elementos de um `Array` respeitam uma dada condição. Para exemplificar, vamos novamente utilizar o cenário dos alunos maiores de idade. Mas para este caso, queremos saber se todos os alunos são maiores de idade. Primeiro, fazemos da forma convencional:

```
var alunos = [
  {nome:'joão', idade: 18},
  {nome:'maria', idade: 20},
  {nome:'pedro', idade: 24}
];

var todosAlunosDeMaior = true;
for(var i = 0; i< alunos.length; i++) {
  if(alunos[i].idade < 18) {
    todosAlunosDeMaior = false;
    break;
  }
}

console.log(todosAlunosDeMaior); // true
```

Iteramos toda a lista procurando por alunos menores de idade. Ao achar um, já encerramos a iteração e retornamos `false`. Agora, observe como podemos simplificar essa lógica usando o `every`:

```
var alunos = [
  {nome:'joão', idade: 18},
  {nome:'maria' idade: 20},
  {nome:'pedro', idade: 24}
];

var todosAlunosDeMaior = alunos.every(function(aluno){
  return aluno.idade > 18;
});

console.log(todosAlunosDeMaior); // true
```

A função itera cada um dos elementos sob a condição de `aluno.idade > 18` e usa o operador lógico `E` (*AND*) em cada um dos retornos. Em outras palavras, caso um dos elementos não satisfaça a condição, o resultado do `every` de imediato será `false`. Caso todos atendam à condição, o `true` é retornado como resultado da função.

3.7 SOME

Se a tarefa é validar se, pelo menos, um dos elementos de um `Array` satisfaz uma dada condição, o `some` é o método perfeito para o trabalho. Imagine que trabalhamos no setor de tecnologia de um aeroporto e precisamos desenvolver um pequeno programa para saber se alguma das malas de um passageiro está acima do limite máximo estabelecido de 30kg. Usando um loop com `for`, o código para tal lógica é semelhante a este:

```
var pesoDasMalas = [12,32,21,29];
var temMalaAcimaDoPeso = false;
for(var i = 0; i < pesoDasMalas.length; i++) {
  if(pesoDasMalas[i] > 30) {
    temMalaAcimaDoPeso = true;
  }
}

console.log(temMalaAcimaDoPeso); // true
```

Mesmo o código não tendo muita complexidade, podemos

torná-lo ainda mais claro, objetivo e enxuto utilizando o `some`.

```
var pesoDasMalas = [12,32,21,29];
var temMalaAcimaDoPeso = pesoDasMalas.some(function(pesoDaMala) {
  return pesoDaMala > 30;
});

console.log(temMalaAcimaDoPeso); // true
```

Para cada peso de mala contida no `pesoDasMalas`, é verificado se ele é superior a 30 kg. Na primeira ocorrência de caso positivo para a condição, a execução do loop é interrompida e o método retorna `true`. Caso contrário, o `Array` todo é percorrido e o método retorna `false` se chegar ao final sem encontrar um registro que satisfaça a condição.

3.8 REDUCE

A função auxiliar `reduce` foi deixada para o final por ser a mais complicada. A ideia por trás dela é pegar todos os valores de um `Array` e condensá-los em um único. Para demonstrar seu funcionamento, vamos mostrar um caso clássico de uso.

Neste exemplo, vamos fazer a soma de todos os elementos de dentro de um `Array`. Como fizemos nos outros, primeiro implementamos uma abordagem mais comum:

```
var numeros = [1,2,3,4,5];

var soma = 0;
for (var i = 0; i < numeros.length; i++) {
  soma += numeros[i];
}

console.log(soma); // 15
```

Aqui não tem segredo. Apenas iteramos a lista com um laço de repetição e usamos a variável `soma`, inicializada em `0`, para acumular o resultado. Agora perceba como temos o efeito

equivalente usando a função `reduce` :

```
var numeros = [1,2,3,4,5];

var soma = 0;
soma = numeros.reduce(function(soma,numero){
  return soma + numero;
}, 0)

console.log(soma); // 15
```

Diferentemente dos outros métodos vistos até agora, o `reduce` aceita dois parâmetros:

- `function(soma,numero){...}` : função de iteração com dois parâmetros;
- `0` : valor inicial.

Para cada iteração, o valor da soma se torna o valor retornado da iteração anterior, sendo que na primeira chamada o valor inicial é o que definimos como o segundo parâmetro da função, neste caso, o número zero. Deu um nó na cabeça, né? Para entender melhor, olhe o diagrama a seguir:

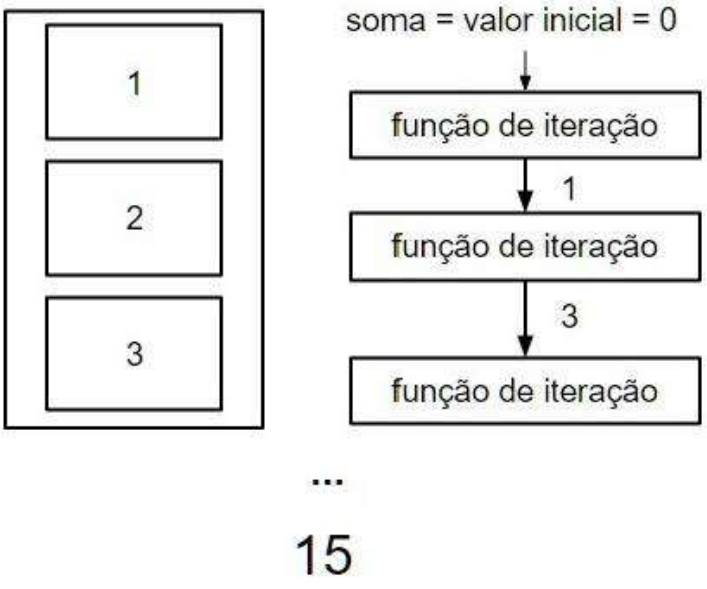

Figura 3.2: Diagrama de funcionamento do reduce

Na primeira iteração, o valor da soma que era zero foi somado ao primeiro valor do `Array`, que era o número 1, dando o total de 1. Esse valor foi acumulado na variável `soma`. Na segunda iteração, o valor do segundo item, o número 2, foi adicionado ao valor da `soma`, que no momento era 1, somando 3. Esse ciclo foi até o último valor, que no exemplo é o 5. Esse valor foi adicionado ao valor da soma que no momento era 10, resultando 15.

Para fixar melhor, veremos um segundo exemplo. Agora temos uma lista de alunos que possuem duas características: `nome` e `idade`. Imagine que queremos uma lista com somente os nomes dos alunos, ignorando a idade. Podemos utilizar o `reduce` para nos ajudar da seguinte maneira:

```
var alunos = [
  {nome:'joão', idade: 10},
  {nome:'josé', idade: 20},
  {nome:'marcos', idade: 30}
];

var nomes = alunos.reduce(function(arrayNomes, aluno) {
  arrayNomes.push(aluno.nome);
  return arrayNomes;
}, []);

console.log(nomes); // ['joão', 'josé', 'marcos']
```

Vamos avaliar o funcionamento deste código. Na lista `alunos`, chamamos o método `reduce`, e nele passamos a função de iteração anônima com dois parâmetros, `arrayNomes` e `aluno`; e um `Array` vazio (`[]`) como valor inicial. Em cada iteração, colocamos o nome do aluno no `Array` de nomes e o retornamos, ou seja, esta variável itera toda a lista e recupera os valores que interessam. De certo modo, condensou o `Array` em um único valor.