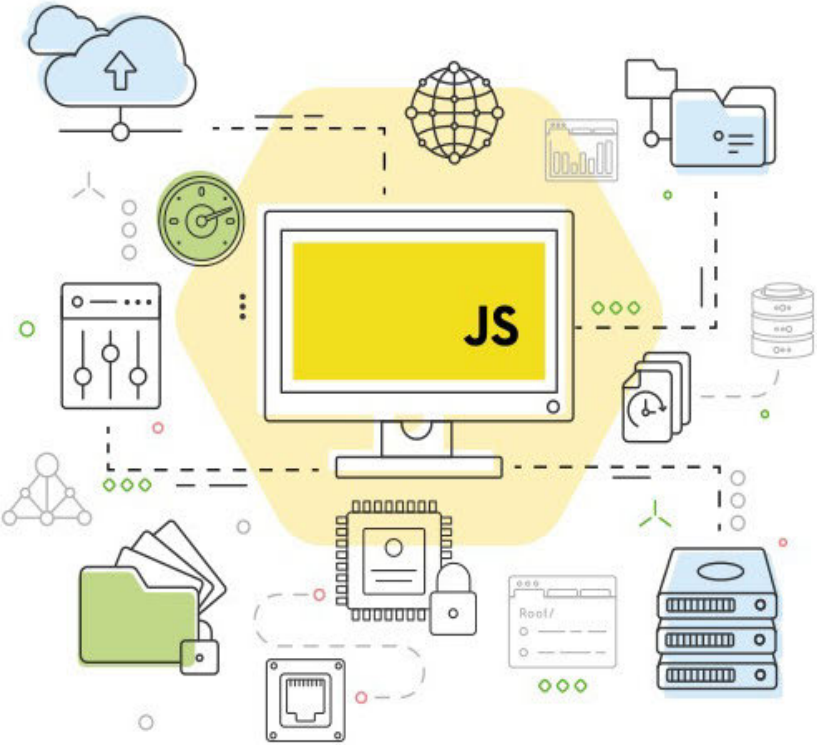
ECMAScript 6

Entre de cabeça no futuro do JavaScript



Casa do Código

DIEGO MARTINS DE PINHO

ISBN

Impresso e PDF: 978-85-5519-258-6

EPUB: 978-85-5519-259-3

MOBI: 978-85-5519-260-9

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

DEDICATÓRIA

A *Camila Aparecida de Souza*. Dedico este trabalho à minha irmã. Crescemos juntos e, durante todos esses anos, ela não somente ajudou a despertar a minha vocação para a área de tecnologia, como também foi minha mentora, me ensinando o pouco que sabia sobre programação e a vida. Infelizmente, não poderei entregar este livro em suas mãos, mas saiba que o dedico inteiramente a você. Obrigado por tudo. Saudades eternas.

Te amo, irmã.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer muito à minha família que sempre me apoiou, motivou e educou a buscar ser melhor sempre. Também gostaria de agradecer à minha namorada, Bruna Freitas, que me cobrou todos os dias a continuar escrevendo até terminar o livro. Sou muito grato aos meus amigos e mentores profissionais, Fabio Nishihara e Marcelo Suzumura, pela confiança no meu trabalho e por todos os ensinamentos dados, tanto na área técnica quanto na de negócios.

Também gostaria de agradecer à editora Casa do Código por me dar esta oportunidade de compartilhar este conhecimento com vocês. Em especial, um agradecimento aos meus editores: Adriano Almeida e Vivian Matsui, que desde o início sempre foram muito atenciosos e prestativos.

E por fim, mas não menos importante, gostaria de agradecer a você que está lendo! Obrigado pela confiança! Espero que este livro o auxilie a se tornar um desenvolvedor ainda melhor!

SOBRE O AUTOR

Figura 1: Diego Pinho

Desenvolvedor de software desde 2013, com experiência na área de *Healthcare* e domínio em tecnologias web de front-end e back-end. Dentre as principais linguagens e frameworks com que trabalha, estão: Java, JavaScript, Spring, Hibernate, Node.js, React.js, entre outras. Tem experiência em metodologias ágeis como o Scrum e Kanban. Hoje atua como Scrum Master em sua equipe.

É bacharel em Ciência da Computação pela Pontifícia Universidade Católica de São Paulo (PUC-SP) e possui MBA em Gerenciamento da Tecnologia da Informação, pela Faculdade de Informática e Administração Paulista (FIAP). É muito criativo e sempre se interessou não somente por tecnologia, mas também por empreendedorismo, processos e gestão de negócios.

Apesar do dia a dia corrido, sempre encontra tempo para continuar aprendendo e para compartilhar o que sabe. Tem uma grande paixão pela educação e gosta muito de ensinar. Escreve artigos sobre tecnologia na internet, faz apresentações em eventos e é entusiasta em programação para jogos, modelagem (2D/3D) e animação.

- **Site:** http://diegopinho.com.br/

PREFÁCIO

O ano era 1995. Estava sendo criada uma nova plataforma de comunicação chamada mIRC (*Internet Relay Chat Client*), e a Microsoft acabava de lançar seu próprio navegador, o Internet Explorer. Ao mesmo tempo, surgia uma nova linguagem, ainda pouco compreendida e, até certo ponto, ridicularizada. Originalmente chamada de Mocha, posteriormente teve seu nome modificado para LiveScript e, por fim, ganhou a imortalidade como JavaScript.

Passadas mais de duas décadas desde o seu surgimento, é praticamente impossível imaginar a internet sem ela. E como é gratificante trabalhar com uma linguagem que evolui dessa maneira! O JavaScript hoje tem um papel fundamental na evolução da web.

O lançamento do ECMAScript 6, a mais recente atualização da especificação, traz uma série de novas features à linguagem. Recursos estes que vão influenciar diretamente no modo como desenvolvemos em JavaScript.

E, assim, o JS continuará sua evolução constante, como tem feito desde sua origem. Em consequência, você precisará evoluir em conjunto. Lembre-se sempre disso, pois a evolução está cada vez mais rápida.

Neste livro, Diego aborda de forma brilhante as principais mudanças que a ECMAScript 6 trouxe à linguagem. Ele não apresenta somente o conceito, mas sim como aplicá-lo na prática e em situações que todos nós vivemos no dia a dia.

Nós dois estivemos no Vale do Silício durante a iMasters DevTrip 2016, visitando algumas das principais empresas que são

referência para nós desenvolvedores. Lá, ficou claro que a grande diferença entre os países não é técnica, mas sim a sua cultura de colaboração.

Lembre-se sempre de que o JavaScript é um código aberto, sem controle centralizado de corporações e de fácil aprendizado. Busque trabalhar em colaboração, integrando-se à comunidade e respeitando as características dessa linguagem criada por Brendan Eich, em 1995.

Conhecimento precisa ser compartilhado, e é exatamente isso que Diego faz neste livro. Parabéns, meu caro!

A você, que segura agora este exemplar, uma excelente leitura. Estude bastante estas páginas e, principalmente, espalhe o seu conhecimento.

Então, vamos começar?

Tiago Baeta

Fundador do iMasters — http://www.imasters.com.br

INTRODUÇÃO

Seja muito bem-vindo a uma nova era na história do JavaScript! Há muitos anos que a linguagem JavaScript não ganhava modificações e funcionalidades novas relevantes, mas isso mudou com a chegada da nova versão da especificação ECMAScript.

Nos últimos cinco anos, o JavaScript ganhou muita força com o "grande boom" do ecossistema Node.js e NPM. A flexibilidade da linguagem tem sido usada a favor não somente do lado do cliente, mas também do lado do servidor. Tudo isso fez com que a comunidade reconhecesse a força da linguagem e finalmente a levasse a sério.

UM BREVE RESUMO DO JAVASCRIPT NA ATUALIDADE

No início da década de 90, começaram a ser desenvolvidas as primeiras aplicações web. A internet estava tomando forma e as páginas começavam a ficar cada vez mais complexas. No entanto, a maior parte dos usuários que tinham acesso à internet utilizava conexões lentíssimas de, no máximo, 28.8 kbps. Você consegue se imaginar usando uma internet dessas hoje em dia? Nem eu.

Como você deve saber, essa velocidade de conexão não estava conseguindo acompanhar a evolução das aplicações web. Elas ficavam maiores e mais complexas, e a velocidade de acesso não conseguia lidar com isso.

Foi nesta época que o JavaScript — também conhecido por nomes como Mocha, LiveScript, JScript e ECMAScript —, uma das linguagens de programação mais famosas do mundo, nasceu. Ela foi criada em 1995 por um engenheiro da Netscape, chamado Brendan

Eich. Originalmente batizada de LiveScript, ela foi lançada pela primeira vez com o Netscape 2, no início do ano de 1996, e prometia ajudar os navegadores dos usuários a acompanhar a evolução das aplicações web.

E para tentar embalar no sucesso na época da linguagem de programação Java, da Sun Microsystems (hoje pertencente a Oracle, que a comprou em 2009), a linguagem mudou o seu nome para JavaScript, mesmo tendo muito pouco em comum com a linguagem Java (e isso causa muita confusão até hoje!). Por este e mais tantos outros motivos, o JavaScript sempre foi conhecido por ser a linguagem de programação mais incompreendida do mundo.

Embora ela tenha sido ridicularizada por muitos anos por ser uma "linguagem de brincadeira", hoje, praticamente após 20 anos do seu lançamento, ela cresceu absurdamente na comunidade de desenvolvedores, tornou-se mais robusta, poderosa e é usada em uma infinidade de aplicações de alto nível, tanto no front-end quanto no back-end das aplicações. Ela é a tecnologia por trás de ferramentas, frameworks e bibliotecas consagradas no mercado, tais como: Angular, Ember, React, Backbone, jQuery, Grunt... A lista é enorme. Atualmente, é essencial que um profissional de TI tenha domínio desta tecnologia, tanto para aplicações web, aplicativos mobile e/ou desktop.

É exatamente para isto que este livro foi escrito. Para que você seja capaz de entender todas as mudanças que vieram com o ECMAScript 6, aprimorar suas habilidades como desenvolvedor e se destacar no mercado de trabalho que hoje é tão concorrido.

A QUEM SE DESTINA ESTE LIVRO

Este livro é destinado aos desenvolvedores web que tenham, pelo menos, conhecimentos básicos da linguagem JavaScript. A

seguir, seguem os principais tópicos que é necessário ter pelo menos um conhecimento básico para melhor aproveitamento do conteúdo:

- Declaração de funções e variáveis
- Estrutura de dados: objetos e arrays
- Funções anônimas e de *callback*
- Escopos
- Herança por prototipagem

Ter domínio desses conceitos, mesmo que seja somente o básico, será essencial. Ao longo do livro, tomei o cuidado de fazer revisões nos principais tópicos da linguagem, mas é imprescindível ter um conhecimento prévio.

Caso você ainda não esteja totalmente seguro sobre os seus conhecimentos nestes tópicos, procure ler alguns livros e artigos, assistir videoaulas, fazer alguns cursos e pequenos experimentos com a linguagem. A internet está recheada de material gratuito de altíssima qualidade sobre o assunto. Dê uma olhada no capítulo de recomendações para ver algumas dicas legais.

Uma vez que você conheça pelo o básico destes tópicos, estará pronto para seguir em frente!

O QUE VOU APRENDER NESTE LIVRO?

Neste livro, abordaremos as principais mudanças que a nova versão da especificação trouxe para a linguagem. Aprenderemos não somente o conceito, mas como aplicá-lo na prática em situações reais. Dentre elas, estão:

- Novas maneiras de iterar objetos e coleções
- Declaração de variáveis com `let` e `const`
- Melhorias em funções com *arrow functions*
- As novas de estruturas de *Map*, *WeakMap*, *Set* e

WeakSet
- Modularização e classes
- Geradores e símbolos
- E muito mais

COMO DEVO ESTUDAR COM ESTE LIVRO?

Este livro foi estruturado de modo que os tópicos apresentados se complementem e se tornem gradualmente mais complexos ao decorrer da leitura. Em todos eles, serão apresentados diversos conceitos, sempre apoiados por códigos contextualizados em casos de uso reais, seguindo as boas práticas. Você notará não somente a evolução dos conceitos, mas também a dos códigos apresentados à medida que a leitura for seguindo.

Como acredito que somente com a prática que aprendemos a programar e fixar o conhecimento, no repositório deste livro no GitHub, você encontrará uma série de exercícios elaborados pensando na prática dos tópicos apresentados. Junto aos exercícios, você também encontrará o gabarito comentado. No final do livro, também há uma série de recomendações de livros, artigos e cursos para que você se aprofunde nos estudos.

Consulte o livro sempre que surgirem dúvidas e entre contato sempre que sentir necessidade. Leia e releia até compreender os conceitos. Não tenha pressa.

Em caso de dúvidas, estarei sempre a disposição. Não deixe de comentar e participar das discussões sobre o livro e os exercícios no site oficial e nos nossos canais de comunicação! O site e o repositório continuarão sendo atualizados com novos exercícios, artigos, notícias e projetos de exemplo. Vamos aprender juntos!

E o mais importante: nunca deixe de praticar!

- **Site oficial:** http://www.entendendoes6.com.br
- **Repositório:** https://github.com/DiegoPinho/entendendo-es6

Sumário

1 ECMAScript x JavaScript **1**

1.1 Implementações da especificação 2

1.2 Breve histórico 3

1.3 Futuro 4

2 Precisamos falar sobre o Babel **6**

2.1 Outras ferramentas 8

3 Métodos auxiliares para Array **10**

3.1 A maneira tradicional de iterar um Array 11

3.2 forEach 12

3.3 map 14

3.4 filter 15

3.5 find 17

3.6 every 18

3.7 some 19

3.8 reduce 20

4 Iteração com iteradores e iteráveis **23**

4.1 Iteradores 23

4.2 Iteráveis 24

4.3 Iteradores e iteráveis na prática 25

5 Iteração com o laço for...of **28**

5.1 Diferenças entre o for...of e for...in 30

5.2 break e continue 30

5.3 Voltando para o Chapéu Seletor 31

6 As novas estruturas de Map e WeakMap **33**

6.1 Map 34

6.2 WeakMap 37

6.3 Administrando uma biblioteca 40

7 Listas sem repetições com Sets e WeakSets **43**

7.1 Set 46

7.2 WeakSet 49

8 Declaração de variáveis com const e let **53**

8.1 Constantes com const 55

8.2 let é o novo var 56

8.3 Qual a diferença, no final das contas? 57

9 Manipulação de textos com template strings **63**

9.1 Template strings simples 65

9.2 Template strings marcado 69

10 Arrow functions **72**

10.1 São menos verbosas 74

10.2 O contexto de execução é diferente 75

11 Melhorias em objetos literais **83**

11.1 Declaração de propriedades 85

11.2 Índices de propriedades computadas 86

11.3 Objetos literais x JSON 87

12 Parâmetros predefinidos em funções **91**

12.1 Atribuindo valores padrões 92
12.2 Valores undefined 94
12.3 Referenciando outros valores padrões 95
12.4 Referenciando variáveis internas 95
12.5 Utilizando funções como valores padrões 96
12.6 Tornando parâmetros obrigatórios 96

13 Parâmetros infinitos com operador Rest 99
13.1 Entenda o que arguments faz 101
13.2 Arguments X Operador Rest 102
13.3 Particularidades do operador Rest 103
13.4 Podemos utilizar em conjunto com parâmetros “fixos” 104

14 Expansão com o operador Spread 106
14.1 Fazendo compras com o Spread 108
14.2 Adicionando itens a um Array 111
14.3 Operador Spread em chamadas de funções 112
14.4 Operador Rest x Operador Spread 114

15 Desestruturamento de Arrays e Objetos 116
15.1 Rotulando propriedades 118
15.2 Desestruturamento de vários objetos 119
15.3 Desestruturamento em retorno de chamadas de métodos 120
15.4 Desestruturamento de Arrays 122
15.5 Desestruturando Arrays — Parte 2 124

16 Modelando com classes 126
16.1 Utilize classes do ES6 130
16.2 Estendendo classes 132
16.3 Hoisting em classes 133
16.4 Declaração por meio de expressões 134

16.5 Métodos estáticos 135
16.6 Atributos privados com WeakMap 135
16.7 Últimas considerações 136

17 Módulos 138
17.1 CommonJS x AMD 140
17.2 Importar e exportar módulos 142
17.3 Exportando classes 146
17.4 Rótulos em módulos 147
17.5 Últimas considerações 148

18 Funções geradoras 150
18.1 O que são funções geradoras? 150
18.2 Iterações com geradores 155
18.3 Entendendo o Symbol.iterator 156
18.4 Delegação de funções geradoras 158

19 Operações assíncronas com promises 161
19.1 O que são promises? 161
19.2 Os estados das promises 164
19.3 O esqueleto de uma promise 165
19.4 Operações assíncronas 167
19.5 Aninhamento de then e catch 168
19.6 Como lidar com erros inesperados 169

20 Metaprogramação com proxies 171
20.1 O que é metaprogramação? 171
20.2 Voltando para os proxies 174
20.3 Validações de inputs com proxies e traps 177
20.4 Desativando um proxy 178
20.5 Últimas considerações 179

21 Um futuro brilhante para o JavaScript 180

22 Recomendações 183

23 Referências 185

CAPÍTULO 1

ECMASCRIPT X JAVASCRIPT

Antes que possamos dar início às explicações e colocar a mão na massa, é muito importante entender corretamente o que é o ECMAScript e qual a sua relação com o JavaScript. O ECMAScript (ES) é a especificação da linguagem de script que o JavaScript implementa. Isto é, a descrição de uma linguagem de script, sendo padronizado pela Ecma International (http://www.ecmascript.org/index.php) — associação criada em 1961 dedicada à padronização de sistemas de informação e comunicação — na especificação ECMA-262.

A especificação é definida pelo ECMA Technical Comittee 39 (TC39), comitê formado por especialistas de grandes companhias da área de tecnologia, tais como: Apple, Google, Microsoft e Mozilla. Este comitê se reúne a cada dois meses, normalmente na Califórnia, para discutir o futuro da especificação. As datas das reuniões, assim como trechos delas, também estão disponíveis online no site oficial.

Somente para se ter uma ideia da grandeza da especificação, a última edição do ECMA-262 (até o momento que este livro foi escrito, estava na 7º edição) possuía 586 páginas. O documento pode ser encontrado no site oficial da Ecma Internacional (http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf).

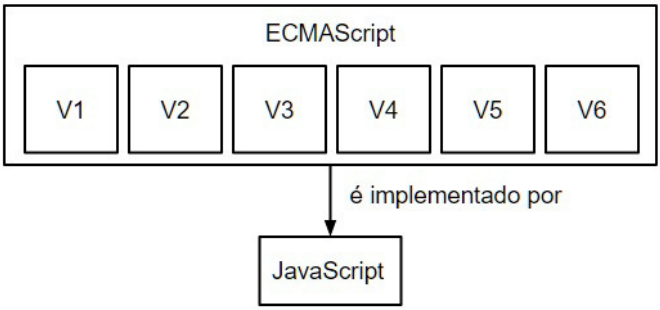


Figura 1.1: A relação entre o ECMAScript e o JavaScript

1.1 IMPLEMENTAÇÕES DA ESPECIFICAÇÃO

O ECMAScript é suportado por uma diversidade de aplicações, principalmente navegadores, em que são implementados pela linguagem JavaScript. Muitas das implementações adicionam extensões específicas a linguagem, como é o caso do JScript da Microsoft. Isso significa que aplicações escritas em uma implementação podem ser incompatíveis com outras. É sempre preciso estar atento a isto.

A tabela a seguir mostra algumas das principais engines e linguagens do mercado que trabalham a especificação.

Implementação	Aplicação	Versão
V8	Google Chrome, Node.js, Opera	6
SpiderMonkey	Firefox, Adobe Acrobat	6
JavaScriptCore (Nitro)	WebKit, Safari	6
JScript 9.0	Internet Explorer	5.1
Nashorn	Java	5.1
ActionScript	Adobe Flash, Adobe Flex e Adobe AIR	4

Muitas engines ainda não dão suporte completo a todas as funcionalidades da ES6. Para se manter informado da

compatibilidade dos navegadores, engines, compiladores e afins que trabalham com ela, podemos usar o ECMAScript Compatibility Table (https://kangax.github.io/compat-table/es6/). Este projeto aberto mantém informações atualizadas sobre a porcentagem da especificação suportada, descriminando todos os itens.

ECMAScript 5 6 2016+ next intl non-standard compatibility table

Sort by Engine types | Show obsolete platforms | Show unstable platforms

V8 | SpiderMonkey | JavaScriptCore | Chakra | Carakan

Minor difference (1 point) | Small feature (2 points) | Medium featu

Feature name	Current browser 97%	Compilers/polyfills 56%	71%	44%	59%	18%	Desktop browsers 11%	83%	95%	86%	91%	92%	92%	94%	97%
		Traceur	Babel + core-js[2]	Closure	Type-Script + core-js	es6-shim	IE 11	Edge 13[3]	Edge 14[3]	FF 45 ESR	FF 48	FF 49	FF 50	FF 51	CH 52+, OP 39+[3]
Optimisation															
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
Syntax															
default function parameters	7/7	4/7	4/7	4/7	5/7	0/7	0/7	0/7	7/7	4/7	4/7	4/7	4/7	6/7	7/7
rest parameters	5/5	4/5	3/5	2/5	4/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
spread (...) operator	15/15	15/15	13/15	12/15	4/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15
object literal extensions	6/6	6/6	6/6	4/6	6/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
for..of loops	9/9	9/9	9/9	6/9	3/9	0/9	0/9	7/9	9/9	7/9	7/9	7/9	7/9	7/9	9/9
octal and binary literals	4/4	2/4	4/4	4/4	4/4	2/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
template literals	5/5	4/5	4/5	3/5	3/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
RegExp "y" and "u" flags	5/5	3/5	3/5	0/5	0/5	0/5	0/5	5/5	5/5	2/5	5/5	5/5	5/5	5/5	5/5
destructuring: declarations	22/22	20/22	21/22	18/22	15/22	0/22	0/22	0/22	22/22	19/22	21/22	21/22	21/22	21/22	22/22
destructuring: assignment	24/24	23/24	24/24	16/24	19/24	0/24	0/24	0/24	24/24	21/24	23/24	23/24	23/24	23/24	24/24
destructuring: parameters	23/23	19/23	20/23	17/23	15/23	0/23	0/23	0/23	23/23	18/23	19/23	19/23	19/23	20/23	23/23
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	0/2	0/2	2/2	2/2	1/2	1/2	1/2	1/2	1/2	2/2
new.target	2/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2

Figura 1.2: ECMAScript Compatibility Table

1.2 BREVE HISTÓRICO

No final de 1995, a Sun Microsystems (adquirida em 2009 pela Oracle por US$ 7 bilhões) e a Netscape Communications Corporation anunciaram o JavaScript, baseado no trabalho de Brendan Eich. Com o sucesso da linguagem para desenvolvimento de páginas web mais interativas, a Microsoft também lançou sua linguagem de script, batizada de JScript, que foi incluída na terceira versão do Internet Explorer (IE) em agosto 1996.

Com o sucesso da linguagem e o surgimento de mais concorrentes, a Netscape submeteu as especificações da sua linguagem para a padronização pela Ecma Internacional. A partir deste ponto, iniciou-se o processo de padronização que gerou a

primeira edição da ECMA-262 em junho de 1997.

Desde então, surgiram seis versões, sendo que a mais atual, objeto de nosso estudo, foi finalizada em 17 de junho de 2015. Veja um histórico resumido de cada uma das versões do ECMAScript:

Edição	Publicação	Principais aspectos
1	Junho, 1997	Primeira versão.
2	Junho, 1998	Pequenas modificações para manter a especificação alinhada com a ISO/IEC 16262.
3	Dezembro, 1999	Foram adicionadas expressões regulares, melhorias no tratamento de strings, definição de erros, tratamento de exceções com try/catch, formatação para output de valores números e outras melhorias.
4	Abandonada	A quarta versão propunha modificações grandes, como a inclusão de classes, modularização, generators e afins. Infelizmente, acabou sendo abandonada devido ao avanço lento e complicações políticas entre os membros da Ecma's Technical Committee 39 (TC39).
5	Dezembro, 2009	Tornou-se mais claro muitas ambiguidades presentes na terceira edição da especificação. Além disso, adicionou algumas novas features, tais como: getters e setters, suporte a JSON e um mecanismo mais robusto de reflexão nas propriedades dos objetos.
5.1	Junho, 2011	Adaptações para que a especificação ECMA ficasse totalmente alinhada com o padrão internacional *ISO/IEC 16262:2011*.
6	Junho, 2015	Vamos ver neste livro cada uma das funcionalidades. ;)

A partir da versão ES6, será adotado o versionamento por ano e não mais por número. É por isso que em muitos lugares encontramos o ES6 como ECMA2015 ou ES2015. São a mesma coisa. Esta nova atitude se deve ao fato da pretensão de termos uma nova atualização da especificação a cada ano.

1.3 FUTURO

Como o objetivo é que, daqui para a frente, tenhamos uma nova

versão da especificação todo ano — para evitar grandes mudanças, como foi da ES5 para a ES6 — a versão ES7, ou ECMA2016, já foi em grande parte definida em junho de 2016 e vai trazer como novidades:

- `Array.prototype.includes` ;
- Operador de exponenciação (`**`);
- Decorators;
- Async-await;
- Propriedades estáticas de classe;
- E muito mais.

Todos os detalhes sobre a ES7 podem ser encontrados na especificação oficial (http://www.ecma-international.org/ecma-262/7.0/). Além disso, o TC39 mantém um repositório no GitHub (https://github.com/tc39/ecma262) com as propostas de funcionalidades e sintaxe consideradas para as versões futuras do ECMAScript.

Capítulo 2

PRECISAMOS FALAR SOBRE O BABEL

Se você já começou a estudar sobre ES6, já deve ter se deparado com este nome: Babel (https://babeljs.io/). Se não for o caso e você nunca ouviu falar, não tem problema. O Babel nada mais é do que um tradutor de JavaScript (o termo normalmente utilizado é *transpiler*).

Com ele, é possível usar os novos recursos do ES6, alguns experimentos do ES7, e até mesmo a sintaxe JSX da biblioteca React nos navegadores atuais, mesmo que não exista compatibilidade. Isso é possível porque o Babel traduz o nosso código JavaScript escrito, usando a nova sintaxe ES6 em código JavaScript equivalente em ES5, a versão adotada pela maioria dos navegadores atualmente. Por isso a maior parte dos projetos que usam ES6 está associada a ele, para garantir compatibilidade.

Figura 2.1: O papel do Babel

Ao decorrer deste livro, utilizaremos o Babel somente como

compilador online (https://babeljs.io/repl/) para escrever código JavaScript ES6 e ver o seu correspondente convertido em JavaScript ES5. Para usá-lo, basta acessar o link indicado.

Você notará que a ferramenta divide a tela em duas partes. Do lado esquerdo, é onde escrevemos o nosso código em ES6, e a ferramenta automaticamente faz a tradução para ES5 do lado direito. Isso nos permite entender como a tradução do código é feita por debaixo dos panos e quais vantagens estamos ganhando ao usarmos a sintaxe ES6.

Além disso, logo abaixo da área do código ES5, existe o espaço que representa o console do navegador. Todas as operações que o usam, como o `console.log`, são exibidas lá.

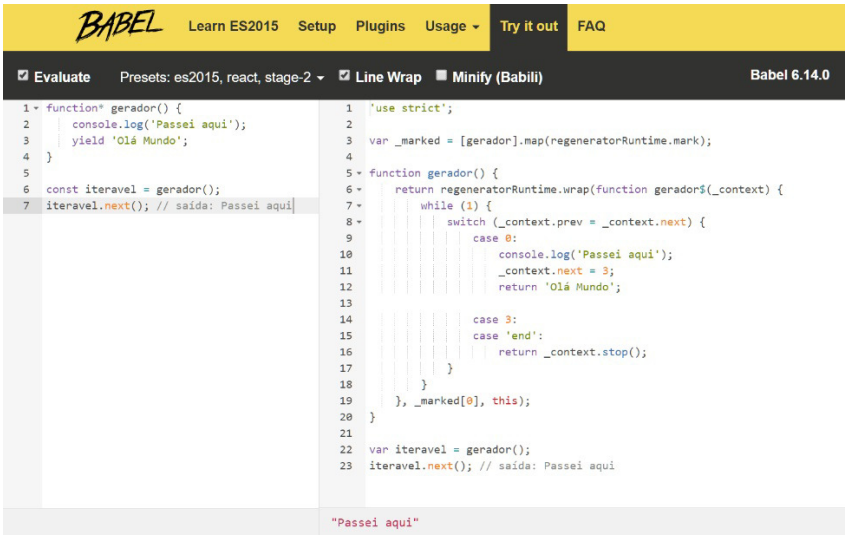


Figura 2.2: Interface do compilador online do Babel

Ao utilizar o compilador do Babel, verifique se a opção `Presets` está marcada com a opção `es2015`. Caso não esteja, marque-a. Isso indica à ferramenta que queremos usar a configuração de tradução de código ES6 para ES5. Certifique-se de

que o `stage-2` também esteja marcado.

Para maiores informações sobre os Presets, acesse: https://babeljs.io/docs/plugins/#presets.

2.1 OUTRAS FERRAMENTAS

Além do compilador online do Babel, existem outras excelentes ferramentas gratuitas disponíveis na internet que podem ser usadas livremente para acompanhar e testar os códigos que serão apresentados neste livro. A única exceção é a do capítulo *Módulos*, pois nele é necessário dividir o código em vários arquivos para ver os efeitos da modularização.

A principal diferença que você notará entre o compilador online do Babel e as demais ferramentas é que elas não apresentam o comparativo entre o código ES5 e ES6. Elas somente o executam.

Dentre as que serão listadas a seguir como sugestão, recomendo a utilização do *repl.it*. Sua interface é bem intuitiva e nos permite salvar, modificar a formatação e compartilhar o nosso código. Além disto, possui uma página de exemplos para iniciantes e também é capaz de interpretar outras linguagens como: Python, Ruby, C/C++, C# e Java. É uma verdadeira mão na roda. Mas reforço que qualquer um dos compiladores listados pode ser usado.

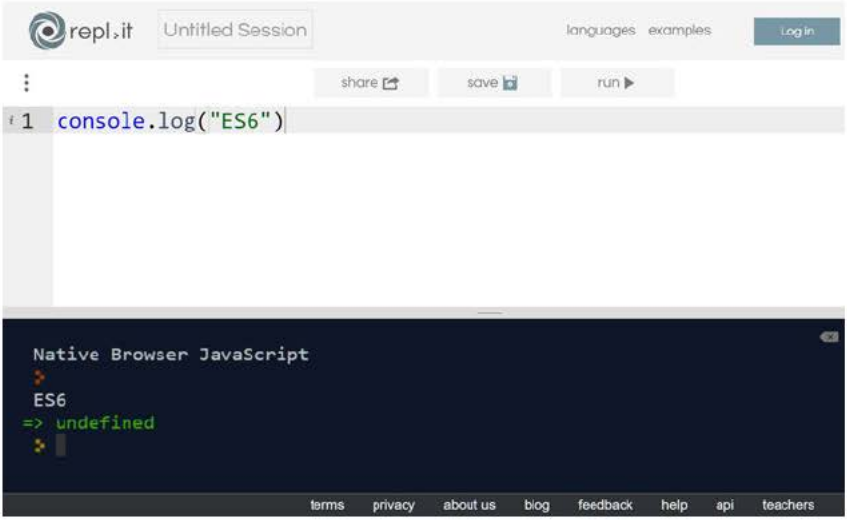


Figura 2.3: Interface do repl.it

Algumas outras ferramentas gratuitas e online são:

- repl.it: https://repl.it/languages/javascript
- JS Bin: https://jsbin.com/
- ES6 Fiddle: http://www.es6fiddle.net/
- ES6 Console: http://es6console.com/

E caso o seu navegador favorito já tenha suporte ao ES6, você também poderá praticar os exemplos direto no console.

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.

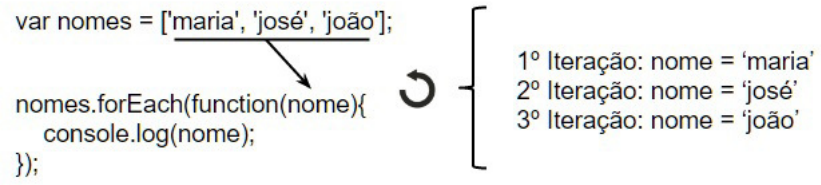


Figura 3.1: Iterando com o forEach

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:

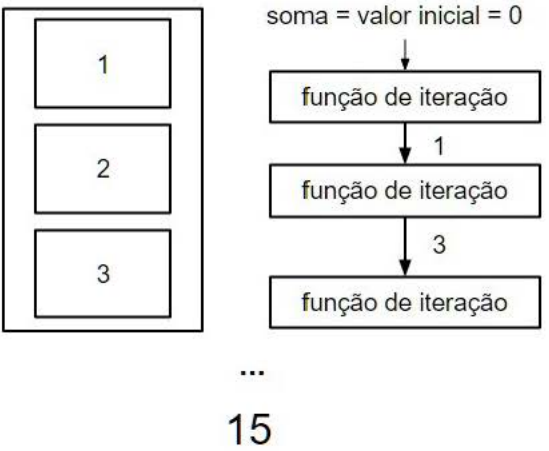


Figura 3.2: Diagrama de funcionamento do reduce

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.

CAPÍTULO 4

ITERAÇÃO COM ITERADORES E ITERÁVEIS

Como vimos nos capítulos anteriores, processar itens em uma coleção é uma operação trivial no dia a dia do desenvolvedor. Não somente no JavaScript, mas na maior parte das linguagens de programação. Entendemos que o JavaScript oferece diversas maneiras de iterar sobre uma coleção, desde do tradicional laço de repetição `for` até os métodos auxiliares de `Array`, como o `map`, `filter` ou `reduce`.

O ES6 introduziu um novo mecanismo para esta tarefa: iteração. O conceito de iteração não é novo e já é utilizado em muitas linguagens de programação como o Java, Python e C# — mas somente agora foi padronizado no JavaScript.

A iteração é definida por dois conceitos centrais: iteradores e iteráveis. Um iterável está ligado com um iterador que define como ele será percorrido. O seu objetivo é prover uma forma de sequencialmente acessar os elementos de um iterável sem expor sua representação interna, retirando a responsabilidade dele de saber como acessar e caminhar sobre sua estrutura. Vamos entender exatamente o que isso quer dizer.

4.1 ITERADORES

Definimos como um iterador um objeto que sabe como acessar,

um a um, os itens de um iterável, enquanto mantém o status da sua posição atual na estrutura. Esses objetos oferecem o método `next`, que retorna o próximo item da estrutura do iterável sempre que invocado.

Na realidade, este método retorna um outro objeto com duas propriedades: `done` e `value`. O `done` é um valor booleano que indica se toda a estrutura foi acessada, enquanto o `value` contém o valor extraído.

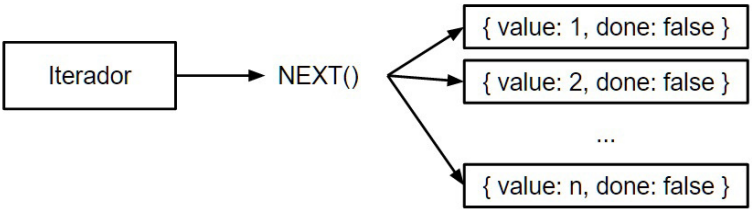


Figura 4.1: Relação entre iterador e iterável

Por exemplo, se tivermos uma coleção com um único número (o número 1) e chamarmos o método `next` uma vez, obteremos este valor:

```
iteravel.next(); // {value: 1, done: false}
```

Se chamamos novamente o `next`, não temos mais valor, pois a coleção inteira já foi percorrida. Entretanto, temos a indicação de que ela foi finalizada na propriedade `done` que retornará `true`:

```
iteravel.next(); // {value: undefined, done: true}
```

4.2 ITERÁVEIS

Um objeto é definido como iterável se ele define explicitamente o seu comportamento de iteração. Para isso, é necessário que ele implemente o seu iterador na propriedade de chave

`Symbol.iterator` (veremos o porquê deste nome com mais detalhes no capítulo *Funções geradoras*). No JavaScript, alguns tipos já são iteráveis por padrão:

- Arrays;
- Strings;
- Maps;
- Sets.

Para estes tipos, podemos obter o seu iterador e usá-lo. Por exemplo, veja o objeto `bruxos` do tipo `Array` do exemplo a seguir:

```
var bruxos = ['Harry Potter', 'Hermione Granger', 'Rony Weasley'];

// obtém o iterador
var iteradorBruxos = bruxos[Symbol.iterator]();

iteradorBruxos.next(); // {value: Harry Potter, done: false}
iteradorBruxos.next(); // {value: Hermione Granger, done: false}
iteradorBruxos.next(); // {value: Rony Weasley, done: false}

iteradorBruxos.next(); // {value: undefined, done: true}
```

Recuperamos o iterador da propriedade `Symbol.iterator` e usamos o seu método `next` para passar por toda a lista.

4.3 ITERADORES E ITERÁVEIS NA PRÁTICA

Como vimos, um objeto iterável está ligado diretamente com um iterador que define como este objeto será percorrido.

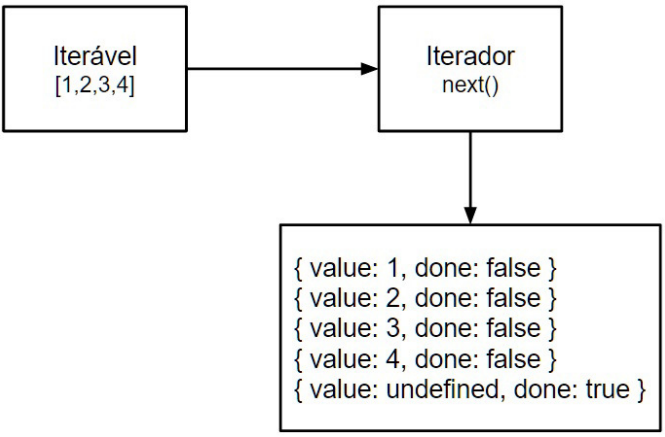


Figura 4.2: Relação entre iterador e iterável

Na prática, a utilização desta estrutura de iteradores e iteráveis é sempre feita pelo laço de repetição `for...of` e geradores. Mas nada impede que usemos a estrutura sem eles.

Por exemplo, vamos supor que queremos fazer uma simulação do Chapéu Seletor de Hogwarts, escola de bruxos da série de livros do Harry Potter, da J.K. Rowling. Para cada bruxo, o Chapéu Seletor deve fazer a seleção de sua casa de acordo com os seus critérios (que não sabemos). Este processo deve ser repetido para todos os bruxos, mesmo não sabendo de antemão quantos serão (sempre haverá pelo menos um, se não a escola vai ficar às moscas).

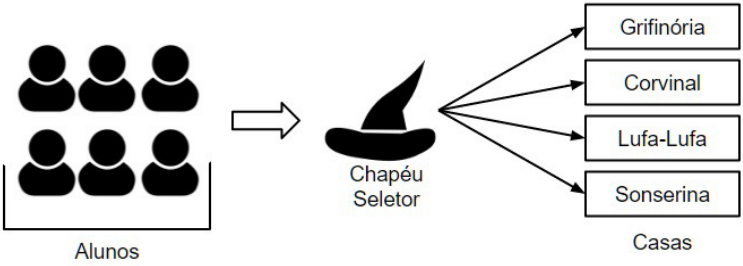


Figura 4.3: Simulação do Chapéu Seletor

Para fazer isso, vamos utilizar iteradores e iteráveis. Assumiremos que todos os bruxos estão em um `Array` chamado `bruxos` e que recebemos essa coleção para iterá-la. Podemos obter o iterador da coleção de bruxos e usar a propriedade `done` em um laço de repetição `do...while`:

```
var iterador = bruxos[Symbol.iterator]();
var done = false;
var proximo = iterador.next();
do {
    var bruxo = proximo.value;
    chapeuSeletor.fazerSelecaoDaCasa(bruxo);
    proximo = iterador.next();
} while (!proximo.done);
```

Imaginando que o método `fazerSelecaoDaCasa` exibe no console o nome do aluno e sua respectiva casa, ao utilizar a nossa lista de bruxos do exemplo anterior neste algoritmo, temos como saída:

```
aluno: Harry Potter | casa: Grifinória
aluno: Hermione Granger | casa: Grifinória
aluno: Rony Weasley | casa: Grifinória
```

Nos capítulos a seguir, aprenderemos como otimizar este código.

FIQUE ATENTO!

Por natureza, iteradores são *Lazy*. Isso significa que ele não sabe de antemão quantos valores vai iterar. É preciso ter muito cuidado, pois ele pode iterar algo infinitamente.

Capítulo 5

ITERAÇÃO COM O LAÇO FOR...OF

Agora que revisamos os principais métodos auxiliares para `Array` e suas aplicações nas mais diversas situações, e também vimos com detalhes como funciona os novos protocolos de iteráveis e iteradores, podemos avançar para a próxima novidade do ES6. Neste capítulo, vamos compreender como funciona um novo tipo de laço de repetição: o laço `for...of`.

Este tipo de laço foi criado para percorrer um objeto se, e somente se, ele for iterável. Seu funcionamento é bem simples. Sua sintaxe é:

```
for (variavel of iteravel) {
    // corpo
}
```

A `variavel` representa uma variável de auxílio que assume valores diferentes a cada iteração, e o `iteravel` é o objeto que será iterado. O caso de uso mais recorrente deste tipo de laço é para passar por todos os valores contidos em um `Array`, `Set` ou um `Map`.

Como veremos mais à frente, ele também é primordial para o uso de geradores. Para contextualizar seu uso, imagine que temos uma sequência de números inteiros de 1 a 5 em uma coleção:

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

Para iterar esta lista `numeros` com o laço `for...of`, é preciso duas coisas de acordo com a sintaxe:

- Declarar uma variável auxiliar que receberá o valor do item da lista a cada iteração;
- Indicar o próprio objeto iterável.

Para este caso, vamos usar uma variável chamada `numero` e utilizá-la para imprimir os valores da lista no console. A implementação fica assim:

```
var numeros = [1,2,3,4,5];
for(var numero of numeros) {
  console.log(numero);
}
```

Ao executar o código, notamos que, para cada iteração, ele imprime o valor correspondente, como esperado:

```
// 1
// 2
// 3
// 4
// 5
```

Agora é que vem a grande sacada. O que acontece por debaixo dos panos é que a estrutura do `for...of` acessa o iterador da estrutura a cada passo da iteração. É por isso que, ao tentar usar o laço `for...of` em objetos que não são iteráveis, obtemos um erro. Como no código seguinte, no qual criamos uma estrutura de dados nova e tentamos iterá-la:

```
var perfilDoFacebook = {
  nome: 'Carlos',
  idade: 22
  // ... outras propriedades
}

for(var dado of perfilDoFacebook){
  console.log(dado);
}
```

```
// TypeError: perfilDoFacebook[Symbol.iterator] is not a function
```

Note que o erro indica que a propriedade `Symbol.iterator` do objeto não é uma função, logo, ele não sabe como fazer para iterar este objeto. Para este tipo de situação, o ideal é utilizar o laço `for...in`.

5.1 DIFERENÇAS ENTRE O FOR...OF E FOR...IN

O laço de iteração `for...in`, disponível no JavaScript já há bastante tempo, funciona de forma diferente. Enquanto ele percorre os nomes dos atributos de um objeto, o laço `for...of` itera somente objetos iteráveis, percorrendo os valores dos atributos. Para entender como ele funciona, vamos usá-lo para iterar o objeto `perfilDoFacebook` do exemplo anterior. Para cada passo, recuperamos o nome do atributo, pegamos diretamente do objeto, e então exibimos no console.

```
var perfilDoFacebook = {
  nome: 'Carlos',
  idade: 22
  // ...
}

for(var propriedade in perfilDoFacebook){
  var info = perfilDoFacebook[propriedade];
  console.log(info);
}
```

Ao executar o código, recebemos os dados do perfil:

```
Carlos
22
```

5.2 BREAK E CONTINUE

Assim como em outros laços de repetição, as palavras reservadas `break` e `continue` também funcionam dentro de laços `for...of`. Usamos o `break` para interromper a execução de um

laço. Nós podemos usá-lo, por exemplo, para colocar uma condição para que um laço seja interrompido caso um número seja maior do que três.

```
var numeros = [1,2,3,4,5];
for(var numero of numeros) {
  if(numero > 3) {
    break;
  }
  console.log(numero);
}

// 1 2 3
```

Já o `continue` usamos para indicar que o laço deve ser continuado, passando imediatamente para o próximo item. Ele é útil, por exemplo, para colocar uma condição no laço para nunca imprimir no console o número dois.

```
var numeros = [1,2,3,4,5];
for(var numero of numeros) {
  if(numero === 2) {
    continue;
  }
  console.log(numero);
}

// 1 3 4 5
```

5.3 VOLTANDO PARA O CHAPÉU SELETOR

No capítulo anterior, fizemos uma simulação do Chapéu Seletor, na qual ele deveria, para cada aluno novo, definir sua casa em Hogwarts. Na ocasião, utilizamos o iterador do `Array` de bruxos e fizemos um controle de repetição com a propriedade `done` dentro da estrutura `do...while`.

Como o laço `for...of` acessa por debaixo dos panos o iterador, podemos refatorar o código e torná-lo bem mais enxuto. Partindo do mesmo princípio que receberemos um `Array`

chamado `bruxos` com cada um dos novos alunos da escola, podemos fazer:

```
for(var bruxo of bruxos) {
  chapeuSeletor.fazerSelecaoDaCasa(bruxo);
}
```

Ao executar o código, teremos o mesmo resultado que no capítulo anterior:

```
aluno: Harry Potter | casa: Grifinória
aluno: Hermione Granger | casa: Grifinória
aluno: Rony Weasley | casa: Grifinória
```

Este laço não somente faz o controle automático, como também já atribui diretamente na variável, a cada iteração, o valor correspondente, sem que nos preocupemos em recuperar o `value`, como fizemos anteriormente.

Parece até bruxaria, mas não é.

Capítulo 6

AS NOVAS ESTRUTURAS DE MAP E WEAKMAP

Mapas são estruturas de dados em que é possível associar uma chave a um valor — como em um dicionário, onde há um significado correspondente para cada palavra. Cada uma das chaves é única e possui apenas um valor associado, mesmo que este se repita em várias chaves.

Quando estamos falando de JavaScript, podemos considerar todo objeto um mapa. Afinal, ambos são constituídos por pares de chave/valor, sendo a chave o nome dos atributos e os valores as funções, objetos e/ou expressões.

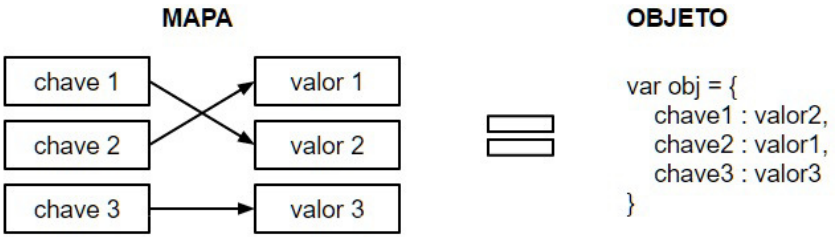


Figura 6.1: Relação Mapas e Objetos do Javascript

Com a chegada do ES6, o JavaScript introduz duas novas estruturas de dados: `Map` e `WeakMap`. Essas estruturas, ao contrário da tradicional concepção de que todos objetos são mapas,

são implementações reais de mapas como estrutura de dados, assim como já estamos acostumados a ver em outras linguagens. Estas novas estruturas nos permitem:

- Adicionar elementos pelo par (chave, valor);
- Remover elementos pela chave;
- Acessar elementos dada uma chave;
- Pesquisar elementos, descobrindo se ele pertece ou não a coleção por meio da chave;
- Indagar sobre atributos, como o número de elementos, por exemplo.

Veremos como cada uma delas funciona.

6.1 MAP

Em um `Map` do JavaScript, qualquer valor (tanto objetos, funções ou valores primitivos) podem ser usados como chave ou valor. Note como conseguimos definir todos estes tipos de valores nos nossos mapas com o método `set`:

```
var map = new Map();
function funcao(){};
var objeto = {};

map.set("string", "sou uma string");
map.set(objeto, "sou um objeto");
map.set(funcao, "sou uma função");
```

Para recuperar os valores do mapa através das chaves, empregamos o método `get`.

```
console.log(map.get("string")); // sou uma string
console.log(map.get(objeto)); // sou um objeto
console.log(map.get(funcao)); // sou uma função
```

Além destes métodos, o `Map` traz algumas outras funções e propriedades interessantes em sua implementação. Para saber

quantos itens um mapa tem, usamos a propriedade `size` :

```
console.log("tamanho: " + map.size); // tamanho: 3
```

Para saber se já existe uma chave específica dentro do mapa, utilizamos o método `has` . Ele retorna um valor booleano: `true` caso exista; `false` caso não exista.

```
console.log(map.has("string")); // true
console.log(map.has("abc")); // false
```

Também podemos remover um registro específico do mapa com o método `delete` , passando como parâmetro a chave do registro que queremos eliminar. Neste exemplo, removemos a chave `string` e usamos o `has` para validar que ele não existe mais.

```
map.delete("string");
console.log(map.has("string")); // false
```

Há também a possibilidade de eliminar todos os registros do mapa usando o método `clear` . Quando o usamos, todos pares de chave/valor são removidos e o mapa fica vazio:

```
map.clear();
console.log("tamanho: " + map.size); // tamanho: 0
```

E por fim, como vimos nos capítulos anteriores, o `Map` é um objeto iterável. Sendo assim, podemos utilizar o laço `for...of` para iterá-los através dos métodos: `keys` , `values` e `entries` . Eles retornam todas as chaves, todos os valores e todas as entradas (par chave/valor), respectivamente.

```
var mapa = new Map();
mapa.set('um', 1);
mapa.set('dois', 2);
mapa.set('três', 3);

for(var chave of mapa.keys()){
  console.log(chave); // um dois três
}

for(var valor of mapa.values()){
```

```
  console.log(valor); // 1 2 3
}

for(var entrada of mapa.entries()){
  console.log(entrada);
}

// saída:
// [ 'um', 1 ]
// [ 'dois', 2 ]
// [ 'três', 3 ]
```

Mapas x Objetos — Quando utilizar cada um deles?

Objetos no JavaScript sempre foram usados como mapas, afinal, ambos são similares no que diz respeito a deixar adicionar chaves/valores, recuperar estes valores através das respectivas chaves e apagá-los também por meio delas. Por convenção, as instâncias de `Map` são úteis somente para coleções, e objetos em geral devem ser usados como registros, com campos e métodos.

Mas nem sempre essa regra é válida. Para nos ajudar em como decidir quando usar uma estrutura ou outra, o site da Mozilla Developer Network (MDN) (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Map) separou algumas perguntas que podemos nos fazer quando estivermos na dúvida:

- As chaves são desconhecidas até o tempo de execução, ou você precisa procurá-las dinamicamente?
- Todos os valores sempre serão do mesmo tipo, e podem ser usados de forma intercambiável?
- Você precisa de chaves que não são Strings?
- Os pares chave/valor são adicionados ou removidos frequentemente?
- Você tem uma quantidade de pares chave/valor arbitrária (de troca fácil)?
- A coleção é iterada?

Se as respostas para as perguntas forem positivas, são sinais de que você provavelmente quer usar uma instância de `Map`. Se em vez disto tivermos um número fixo de chaves com as quais trabalha individualmente e faz diferenciação entre o seu uso, queremos um objeto.

6.2 WEAKMAP

Um `WeakMap` é uma coleção de pares de chave/valor na qual as chaves só podem ser objetos. As referências do objetos nas chaves são fracamente mantidas. Isso significa que eles não estão previnidos de serem coletados pelo *Garbage Collector* caso não existir nenhuma outra referência para o objeto em memória.

Para exemplificar, criaremos uma `WeakMap` onde armazenaremos dois elementos de uma página HTML:

```
var weakMap = new WeakMap();
var elemento1 = window;
var elemento2 = document.querySelector('body');

weakMap.set(elemento1, 'sou o elemento1');
weakMap.set(elemento2, 'sou o elemento2');
```

Neste momento, quando recuperamos os valores por meio das chaves, obteremos o resultado esperado:

```
console.log(weakMap.get(elemento1));
console.log(weakMap.get(elemento2));

// saída
// sou o elemento1
// sou o elemento2
```

Agora, quando removemos todas as referências do `elemento2`, vemos que não é possível adquirir o valor com a chave no `WeakMap`:

```
elemento2.parentNode.removeChild(elemento2);
elemento2 = null; // removendo referência local
```

```
console.log(weakMap.get(elemento2)); // undefined
```

> **O que é o Garbage Collector?**
>
> É o termo usado para descrever o processo automatizado de gerencimento de memória. Este processo recupera as partes da memória que não estão sendo mais usadas, evitando que ela se esgote com o tempo e seja liberada para outros processos.

Como toda chave do `WeakMap` necessariamente precisa ser um objeto, se tentamos utilizar qualquer outro tipo de valor no lugar, tomamos um erro:

```
var weakMap = new WeakMap();
function funcao(){};
var objeto = {};

// TypeError: Invalid value used as weak map key
weakMap.set("string", "isso é uma string");

weakMap.set(funcao, "isso é uma função");
weakMap.set(objeto, "isso é um objeto");
```

Além disso, dos métodos que vimos em `Map`, somente temos quatro deles disponíveis aqui: `delete`, `has`, `get` e `set`. Não temos como limpar todos os dados de uma vez (método `clear`) e nem ter uma visão geral do conteúdo do `WeakMap` (método `entries`) por causa da natureza fraca das ligações.

Onde usar WeakMaps?

Para ser honesto, há poucas circunstâncias no dia a dia em que aplicaremos o `WeakMap`. Para a maior parte dos problemas, a implementação de `Map` será o suficiente. Quando usamos um `WeakMap`, temos a vantagem de que ele nos permite armazenar

dados em um objeto em particular na nossa aplicação e, quando este objeto é destruído, os dados também são destruídos e a memória fica livre para atuar em novos processos. Isso é bacana, pois nos dá a segurança de que:

- Não haverá problemas de vazamento de memória (*memory leak*) na aplicação;
- Poderemos manter dados privados dentro da aplicação, sem expor o que não for necessário.

Temos como usar utilizar esta estrutura, por exemplo, para restringir o acesso do usuário a certas propriedades de um objeto, como na construção de uma API, na qual queremos dar poder para o desenvolvedor para usar nosso código, mas não queremos que ele saia espiando e usando o que não deve. A abordagem convencional para "proteger" uma propriedade de acesso indevido é usar um `_` (*underscore*) como prefixo no atributo. Veja o código a seguir, no qual queremos que `nome` seja um atributo privado do objeto `Pessoa`.

```
function Pessoa(nome) {
    this._nome = nome;
}

Pessoa.prototype.getNome = function() {
    return this._nome;
};
```

Nesta abordagem, bem utilizada pelos desenvolvedores no JavaScript ES5, atribuímos a propriedade `nome` no construtor do objeto `Pessoa` e, por prototipagem, atribuímos a função `getNome`, de modo que todas as instâncias de `Pessoa` usam o mesmo método para recuperar o valor da propriedade `nome`. Entretanto, esta abordagem não protege a propriedade, já que ela continua acessível:

```
var roberto = new Pessoa('Roberto');
console.log(roberto.getNome()); // Roberto
```

```
console.log(roberto._nome); // Roberto
```

Ao usar um `WeakMap`, conseguimos esconder a propriedade que guarda o valor e oferecer somente um método para recuperá-lo.

```
var Pessoa = (function() {

    var dadosPrivados = new WeakMap();

    function Pessoa(nome) {
        dadosPrivados.set(this,{ nome: nome });
    }

    Pessoa.prototype.getNome = function() {
        return dadosPrivados.get(this).nome;
    };

    return Pessoa;
}());

var rafael = new Pessoa('Rafael');
console.log(rafael.getNome()); // Rafael
console.log(rafael.nome); // undefined
```

No capítulo *Modelando com classes*, veremos como usar este mesmo tratamento para criar dados privados dentro de uma classe.

6.3 ADMINISTRANDO UMA BIBLIOTECA

Imagine que fomos contratados para criar um pequeno sistema web para administrar uma biblioteca escolar. A escola Dantas Júnior cresceu bastante nos últimos anos e, junto com ela, o seu acervo de livros.

Como a quantidade é muito grande, as pessoas começaram a se perder na biblioteca, pois nunca sabiam onde os livros estavam guardados. O requisito é que o sistema permita que, dado o título de um livro, ele revele sua localização nas estantes.

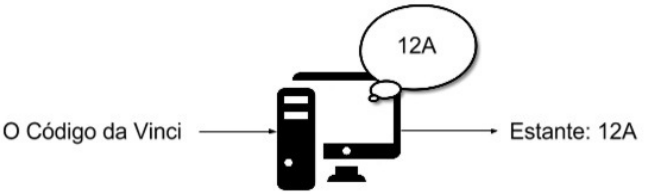


Figura 6.2: Simulação do sistema web da biblioteca

Para resolver este problema, no cadastro dos livros no sistema, podemos implementar um mapa, em que associamos a localização na estante (chave/valor) para cada título. Vamos supor que a carga inicial seja feita por outro sistema terceiro e dada para nós na lista `livros`.

Cada livro usa a estrutura de `WeakMap` para guardar o valor dos seus atributos, sendo que só são oferecidos alguns métodos em sua API para que possamos utilizar, e só nos interessa dois deles: `getEstante` e `getTitulo`. Dado este cenário, podemos fazer:

```
var estantes = new Map();
for(var livro of livros) {
    estantes.set(livro.getTitulo(), livro.getEstante());
}
```

Criamos um mapa chamado `estantes` e colocamos todos os livros que recebemos da API hipotética na estante, usando o título como chave e a estante como valor. Agora já temos a carga inicial. O próximo passo é pensar em como recuperar estes livros.

Pensando em uma tela de busca, o usuário vai inserir o nome do livro que ele quer localizar, e então podemos recuperar sua localização no mapa com o método `get`. Vamos colocar essa lógica em um método chamado `getLocalizacaoDoLivro`. O metodo vai receber um parâmetro chamado `tituloDoLivro`, que será o nome do livro que o usuário deseja localizar. A implementação fica simples:

```
function getLocalizacaoDoLivro(tituloDoLivro) {
    var estante = estantes.get(tituloDoLivro);
    if(estante === undefined) {
        return 'Livro não encontrado no acervo!';
    }

    return estantes.get(tituloDoLivro);
}
```

Quando o livro não é encontrado no mapa, a mensagem `Livro não encontrado no acervo!` é devolvida para o usuário. Agora basta chamar este método na tela de busca e ninguém mais perderá um livro na biblioteca!

Capítulo 7

LISTAS SEM REPETIÇÕES COM SETS E WEAKSETS

Às vezes, necessitamos tratar coleções de itens únicos, ou seja, itens que não repetem. Estes itens podem ser qualquer coisa: números, strings, objetos ou mesmo funções. Na prática, isso significa que precisamos criar listas em que só podemos adicionar um item específico uma única vez. Se tentarmos adicioná-lo mais de uma vez, ela deve ser inteligente o suficiente para saber que aquele item já está lá e não o adicionar novamente.

No JavaScript na versão ES5, não temos uma implementação nativa deste tipo de estrutura com essa inteligência. Esta estrutura é chamada de `Set` e já está presente em diversas linguagens.

Como não temos o `Set`, para resolver nossos problemas, precisamos criar nossa própria implementação. Temos de ter algum jeito de saber se um dado elemento já está na lista toda vez que adicionamos algo nela. Como tudo no mundo da programação, existem inúmeras formas de se fazer isso. Uma das maneiras imediatas é utilizando o método `indexOf` toda vez que precisamos adicionar um elemento na lista, para verificar se o elemento já existe nela.

Para contextualizar este exemplo, pense que estamos implementando um sistema online de loteria. Neste sistema, os usuários podem fazer o seu jogo eletronicamente da Mega-Sena,

Lotofácil, Quina, Lotomania e outros jogos. Em todos eles, o jogador insere em quais números ele quer apostar. Como os números não podem se repetir em nenhum dos jogos, podemos usar um `Set` para garantir isto.

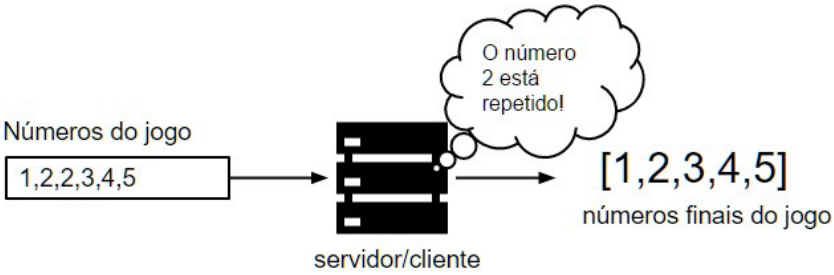


Figura 7.1: O Set como nosso aliado para evitar repetições

Primeiro, iniciamos nossa implementação criando um método chamado `add` que será responsável por adicionar itens na lista. Neste objeto que estamos criando, utilizamos então uma estrutura de `Array` interna que armazenará todos os valores que vamos inserir.

No método `add`, usamos esta estrutura interna para fazer a chamada do `indexOf` que busca pelo objeto alvo dentro no nosso `Set`. Se a função retorna o valor `-1`, significa que não existe nenhuma instância do objeto que queremos adicionar na nossa coleção, logo, podemos adicionar. Caso contrário, não. Veja a implementação deste algoritmo:

```
function Set() {
    var array = [];
    this.add = function(valor) {
        if(array.indexOf(valor) === -1) {
            array.push(valor);
        }
    }
}
```

Para facilitar nossos testes, também criamos o método `mostrarValores` que é responsável por mostrar no console todos os valores contidos no `Set`.

```
function Set() {
    var array = [];
    this.add = function(valor) {
        if(array.indexOf(valor) === -1) {
            array.push(valor);
        }
    },

    this.mostrarValores = function() {
        console.log(array);
    }
}
```

Agora que já temos estes dois métodos, já podemos fazer alguns testes com a nossa estrutura. Vamos começar tentando incluir alguns números. Para validar a lógica contra repetição de itens, adicionaremos o número 2 (dois) duas vezes no nosso `Set`.

```
var set = new Set();
set.add(2);
set.add(1);
set.add(2);

set.mostrarValores(); // [ 2, 1 ]
```

Nossa implementação, apesar de incompleta, já funciona. Entretanto, repare que é possivelmente uma abordagem bem lenta para listas muito grandes. Além disso, se quisermos implementar um `Set` totalmente funcional, temos de gerar muitas outras linhas de código para essa estrutura. Adicionar no mínimo métodos para:

- Remover um único elemento específico;
- Remover todos os elementos;
- Verificar se dado elemento já existe;
- Quantificar o número de elementos.

Com a chegada do ES6, temos uma estrutura de `Set`

implementada nativamente no JavaScript. Essa implementação já contém todos os métodos que sugerimos anteriormente e muito mais. Além disso, análogo ao `WeakMap`, também temos o `WeakSet`. Vamos ver com detalhes como estas estruturas funcionam.

7.1 SET

O `Set` é uma estrutura de dados que nos permite ter listas com valores que nunca se duplicam e que mantém a ordem de inserção dos seus itens. Podemos trocar a implementação que fizemos no tópico anterior para usar a nova implementação nativa do JavaScript ES6.

O código fica basicamente o mesmo, com a diferença do método `mostrarValores`, que não existe na implementação nativa. Por isso, vamos remover este método e usar uma simples iteração por laço `for...of` para ver seus valores. Como o `Set` também é um objeto iterável, isso não é problema:

```
var set = new Set();
set.add(2);
set.add(1);
set.add(2);

for (const valor of set) {
    console.log(valor); // 2, 1
}
```

O construtor do `Set` nos permite passar quais valores queremos com que ele seja inicializado. Para isso, basta passar os valores desejados dentro de um `Array`. Usando este construtor, já somos capazes de eliminar algumas linhas de código, veja:

```
var set = new Set([2,1,2]);
for (const valor of set) {
    console.log(valor); // 2, 1
}
```

Repare em um detalhe interessante. Mesmo colocando propositalmente o valor 2 (dois) duplicadamente na lista, o `Set` se encarregou de se livrar da cópia, mesmo sem dar nenhum indício disto para nós. Isso é uma mão na roda para quando precisamos lidar, por exemplo, com entradas do usuário no sistema, onde é tudo imprevisível.

Para mostrar as demais operações possíveis com `Set`, usaremos mais um sistema hipotético. Imagine agora que trabalhamos em uma empresa responsável por um sistema de streaming de músicas corrente ao Spotify, mas que tem uma particularidade bem interessante: é focado para programadores. Um serviço de streamimg de músicas para programadores.

Uma das features mais bacanas deste sistema é a criação de listas de músicas personalizadas. O usuário é livre para adicionar as músicas que ele quiser na lista, no entanto, ela só pode ser incluída uma única vez em cada lista.

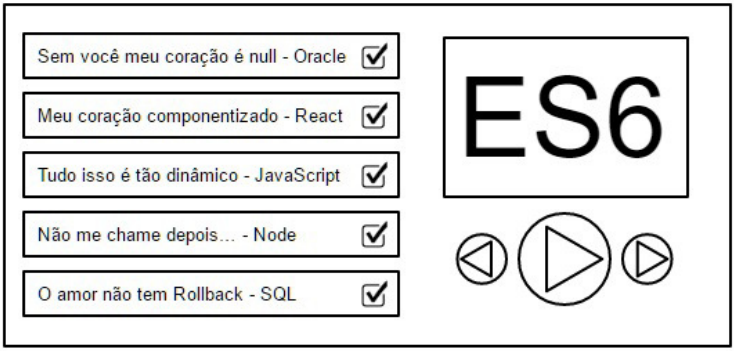


Figura 7.2: Interface do serviço de música para programadores

Para lidar com todas as operações de lista de músicas no sistema, utilizaremos os métodos que o `Set` nos oferece. Para facilitar o entendimento dos nossos exemplos, vamos representar as nossas músicas somente pelo seu título. Faremos uma demonstração com

objetos literais mais complexos mais à frente.

Para adicionar uma música a lista personalizada do usuário, usamos o método `add` visto no exemplo anterior. Este método aceita apenas um parâmetro: o objeto que pretendemos inserir no `Set`.

```
var musicas = new Set();
musicas.add('musica1');

for(var musica of musicas) {
    console.log(musica); // musica1
}
```

Assim como oferecemos a opção de adicionar uma música, também precisamos oferecer a opção de deletar uma música da lista. Para isso, usamos o método `delete`. Este aceita como parâmetro o elemento que queremos remover da lista. Caso este elemento não esteja na lista, nada acontece:

```
var musicas = new Set(['musica1','musica2']);
musicas.delete('musica1');

for(var musica of musicas) {
    console.log(musica); // musica2
}
```

Mas se o usuário quiser deletar todas as músicas da sua lista de uma vez só, podemos criar um botão que faça isso utilizando o método `clear`. Este método remove todos os itens do nosso `Set`, deixando-o inteiramente vazio:

```
var musicas = new Set([
    'musica1','musica2', 'musica3'
]);

musicas.clear();

for(var musica of musicas) {
    console.log(musica); // nada
}
```

Outro ponto importante da aplicação é mostrar para o usuário quando ele estiver em uma música, se ela já foi incluída na lista ou não. Para isto, usamos o método `has`. Ele sempre retorna um valor booleano: `true` em caso positivo, e `false` para negativo.

```
var musicas = new Set(['musica1']);
if(musicas.has('musica1')) {
    console.log('já está na lista!');
}

// já está na lista!
```

Por fim, o usuário também pode querer saber quantas músicas ele já adicionou em sua lista. Podemos fazer isso facilmente com a propriedade `size` do `Set`.

```
var musicas = new Set([
    'musica1','musica2', 'musica3'
]);

var qtdMusicas = musica.size;
console.log("Há " + qtdMusicas + " músicas na lista");
// Há 3 músicas na lista
```

Pronto! Já temos nossa funcionalidade de lista de músicas praticamente implementada usando apenas o `Set`.

7.2 WEAKSET

O `WeakSet` é um `Set` que não previne os seus elementos de serem coletados pelo *Garbage Collector*. Uma vez que o elemento não existe mais e seja identificado pelo coletor para ser coletado, ele também é automaticamente removido do `WeakSet`. Além disso, há três restrições importantes nesta estrutura que o diferem do `Set`:

- **Só é possível adicionar objetos:** valores de tipos primitivos como números e strings não são aceitos;
- **O `WeakSet` não é iterável:** isso significa que não podemos usar laços de repetição como o `for...of`

nele;

- **Não há como remover todos os elementos de uma vez:** esta estrutura não implementa o método `clear`, pois a lista é mantida "automaticamente" pelo *Garbage Collector*.

Quando dizemos que a lista se mantém automaticamente, isso significa que não é preciso se preocupar com vazamentos de memória, pois a lista nunca vai conter uma referência para algo que não existe mais. Vamos ver como funciona com um exemplo e, para tal, retornaremos a funcionalidade de lista de músicas do nosso streaming de músicas.

A primeira mudança que precisamos fazer para usar uma estrutura de `WeakSet` nele é que não podemos mais representar nossas músicas apenas pelo título; temos de atribuí-la a um objeto. Então, vamos fazer isso guardando como propriedade o título e o autor:

```
var musica1 = {
    titulo: 'O amor não tem rollback',
    autor: 'SQL'
}
```

Agora que fizemos isso, criamos um `WeakSet` para colocar nossa música:

```
var musica1 = {
    titulo: 'O amor não tem rollback',
    autor: 'SQL'
}

var musicas = new WeakSet([musica1]);
console.log(musicas);
// WeakSet {Object {titulo: "O amor não tem rollback", autor: "SQL
"}}
```

Agora, se apagarmos a referência da `musica1`, ela eventualmente será coletada pelo *Garbage Collector*. Só precisamos

estar em alerta para o fato de que o coletor funciona de forma diferente em cada navegador e não pode ser forçado. Como ele não pode ser forçado, precisamos que ele aja por conta própria.

Assim, precisamos fazer a demonstração em duas partes. Primeiro, vamos remover a referência da música atribuindo o valor `null` a ela.

```
var musica1 = {
    titulo: 'O amor não tem rollback',
    autor: 'SQL'
}

var musicas = new Set([musica1]);
console.log(musicas);

musica1 = null;
```

Se fizemos o `console.log(musicas)` logo de imediato, é possível que não tenha dado tempo para o coletor ter identificado a referência nula e ter removido o `musica1`. Isso significa que a lista ainda estará guardando a referência para o objeto que não existe mais. Por isso, aguarde alguns poucos segundos, e então execute o `console.log`:

```
console.log(musicas); // {}
```

Note que a lista se atualizou "automaticamente". O `WeakSet` verifica quais dos seus elementos não possui mais nenhuma referência válida, e então o remove.

Este é o poder do `WeakSet`. Há poucas circunstâncias nas quais ele poderá ser útil no dia a dia. Existem muitas discussões em fóruns e blogs de desenvolvimento sobre as possíveis utilidades da estrutura de `WeakSet`. Um dos casos de uso mais interessante é o de garantir que certo método ou propriedade pertence a um objeto específico e não a todas as instâncias do mesmo tipo. Mas no uso geral, sempre que você tiver preocupação com vazamento de memória, o `WeakSet` estará a seu dispor.

Capítulo 8

DECLARAÇÃO DE VARIÁVEIS COM CONST E LET

Uma das primeiras coisas que aprendemos quando estudamos JavaScript é que ela é uma linguagem fracamente tipada, ao contrário de linguagens como C++, Cobol, Fortran e Java, que são fortemente tipadas. Na prática, isso siginifica que as variáveis não têm nenhum tipo predeterminado. Em vez disso, o seu tipo foi definido pelo seu valor.

Para definir qualquer tipo de variável no JavaScript, utilizamos a palavra reservada `var`. Elas podem ser objetos literais, números, caracteres, strings, listas ou funções:

```
// exemplos
var objeto = {}; // objeto
var numero = 1; // numero
var nome = "Chaves"; // string
var lista = [1,2,3]; // lista
```

É POSSÍVEL ESCREVER JAVASCRIPT TIPADO?

Hoje já existem tecnologias que nos permitem escrever JavaScript tipado. A solução mais famosa é o TypeScript, tecnologia de código aberto adotada pelo Angular 2 como linguagem oficial. Essa linguagem, como o próprio site oficial descreve, nos permite escrever JavaScript utilizando uma estrutura fortemente tipada e ter este código compilado para JavaScript puro, para qualquer navegador, host e sistema operacional.

Site oficial: https://www.typescriptlang.org/.

O fato de ser uma linguagem pouco tipada nos permite fazer uma série de coisas que não são possíveis em linguagens tipadas. Uma delas é realizar operações entre tipos diferentes sem causar nenhuma exceção. Podemos, por exemplo, concatenar uma `String` a um inteiro (`int`):

```
var texto = 'texto';
var idade = 10;

console.log(texto + idade); // texto10
```

O interpretador JavaScript identifica os tipos, converte um deles no outro e, em seguida, realiza a operação. Fazendo um comparativo com o Java, o código anterior equivalente é bem mais chato de se escrever. Temos de converter explicitamente as variáveis para um tipo comum antes de fazer operações com elas.

```
// imports...

String texto = "texto";
int idade = 10;

System.out.println(texto + String.valueOf(idade)); // texto10
```

Mas mesmo com essas vantagens, ainda há muitos problemas com variáveis no ES5, como veremos ao decorrer do capítulo. Com a chegada do ES6, o modo de trabalhar com variáveis mudou. O JavaScript continua sem tipo como era antes, mas até então, usávamos somente o `var` para declarar as nossas variáveis, independente do seu tipo e contexto. Esta regra mudou: agora temos duas novas palavras reservadas que o substituem: `const` e `let`.

Vamos entender como cada uma delas funciona e quais os problemas que elas resolvem.

8.1 CONSTANTES COM CONST

Usaremos o `const` nas variáveis que não esperamos que mudem de valor com o decorrer do tempo/execução do programa dentro de um escopo. Por exemplo, se estivermos tratando a data de nascimento de uma pessoa em um registro de um site de compras, podemos armazená-la em uma variável utilizando o `const`, já que a data de nascimento é um valor que não vai se alterar:

```
const dataNascimento = '21/07/1992';
```

Como o objetivo é que variáveis declaradas com `const` não mudem de valor, se tentarmos burlar isso atribuindo um outro valor para a variável, levamos um erro, como pode ser visto no exemplo a seguir:

```
const dataNascimento = '21/07/1992';
dataNascimento = '25/08/1996'; // tentando atribuir novo valor

// TypeError: Assignment to constant variable.
```

O erro indica que estamos tentando atribuir um valor novo a uma constante. Todavia, é necessário ter em mente que um `const` não se trata de uma variável constante, mas uma referência

constante. Em termos práticos, isso significa que o valor não é imutável, é possível adicionar e remover propriedades desta variável. Como no próximo exemplo, no qual declaramos um objeto `pessoa` com uma propriedade `nome` e adicionamos uma outra propriedade `idade` a ela:

```
const pessoa = {nome: 'joão'};
pessoa.idade = 12;

console.log(pessoa); // {"nome":"joão", "idade": 12}
```

A referência ao objeto continua a mesma, por isso não recebemos nenhum erro. Mas se tentamos atribuir um outro objeto a esta variável, novamente não conseguimos:

```
const pessoa = {nome: 'joão'};
const novaPessoa = {nome: 'maria'};

pessoa = novaPessoa;

// TypeError: Assignment to constant variable.
```

8.2 LET É O NOVO VAR

Podemos considerar o `let` como o verdadeiro substituto do `var`. Idealmente ele deve ser usado para declarar variáveis que esperamos que mudem com o valor e com o tempo/execução do programa. Por exemplo, usamos o `let` para atribuir o valor da soma de dois números:

```
let resultado = 0;
resultado = soma(2,3);
console.log(resultado); // 5
```

Entretanto, ao contrário do que acontecia com o `var`, se uma mesma variável, dentro de um mesmo escopo, é declarada duas vezes, tomamos erro de sintaxe. Isso acontece tanto para o `let` quanto para o `const`. Veja:

```
var id = 1;
```

```
var id = 2;
console.log(id); // 2

let id = 1;
const id = 1;
// SyntaxError: Identifier 'id' has already been declared
```

Usando o `var`, não temos problema. O interpretador percebe que se trata da mesma variável e altera o seu valor. Mas quando tentamos usar a mesma variável com o `const` ou `let`, não conseguimos. É preciso declarar com outro nome.

8.3 QUAL A DIFERENÇA, NO FINAL DAS CONTAS?

Somente com esta explicação, pode não parecer o suficiente justificar a mudança do `var` para `const` e `let`. E você está certo. Há muitas outras razões que motivam a criação do `const` e `let`. Vamos vê-las.

Diferença de escopo entre var e let/const

Um dos grandes problemas com a utilização do `var` no ES5 é que seu comportamento pode ser bem confuso. Observe atentamente o código:

```
var mensagem = 'olá';
{
  var mensagem = 'adeus';
}
console.log(mensagem);
```

Se eu lhe perguntar qual o valor da variável `mensagem` que é exibido no final da execução, é bem provável que você responda: "olá". E seu pensamento faz sentido, afinal, a segunda variável `mensagem` está dentro de um bloco, e isso provavelmente não deve interferir no valor da primeira variável, certo? Errado.

A mensagem "adeus" que aparece no console. Isso porque, nas duas declarações, estamos tratando da mesma variável `mensagem`, na qual o valor está sendo redefinido no bloco entre `{}` (chaves). Esta conduta se dá ao fato de que as variáveis no ES5 possuem o que é chamado de escopo de função.

Isso significa que as funções são os contextos delimitantes aos quais valores e expressões estão associados. Neste trecho, como a segunda declaração da variável não está delimitada por nenhuma função, mas somente um bloco, ela sobrescreve o valor da primeira.

Para mudar este efeito, precisamos criar uma função que delimita o contexto da segunda variável `mensagem`.

```
var mensagem = 'olá';
function mensagem() {
  var mensagem = 'adeus';
}
console.log(mensagem); // olá
```

Agora que temos a função `mensagem`, as variáveis que são declaradas dentro dela não afetam as que já existem fora. Entretanto, se usarmos um loop de repetição `for` ou qualquer tipo de estrutura que utilize bloco, isto não funcionaria, já que teríamos o mesmo comportamento do primeiro exemplo.

Para evitar este problema, agora no ES6 as variáveis possuem escopo de bloco. Vamos ver a diferença entre os escopos com um pequeno experimento. Aqui, vamos iterar os números de 1 a 5 e, para cada iteração, inserir uma função que loga este valor dentro de um `Array`. Isso nos ajudará a perceber o comportamento das variáveis.

Primeiro, iteramos com o `var` dentro do escopo do laço de repetição `for`:

```
const arrayVar = [];
for(var i = 1; i < 5; i++) {
```

```
    arrayVar.push(function () {
          console.log(i);
    });
}
```

Agora a mesma coisa, mas utilizando o `let` :

```
const arrayLet = [];
for(let i = 1; i < 5; i++) {
    arrayLet.push(function () {
          console.log(i);
    });
}
```

Agora, iteramos o `arrayVar` e o `arrayLet` usando o `forEach` para analisar os resultados:

```
arrayVar.forEach(function (funcao) {
   funcao(); // 5 5 5 5 5
});

arrayLet.forEach(function (funcao) {
  funcao();  // 1 2 3 4 5
});
```

Resultado interessante, não? Apesar de parecer que ambos teriam o mesmo resultado (imprimir de 1 a 5), precisamos levar em conta o que acabamos de ver sobre a diferença nos escopos (função *x* bloco). Como as variáveis declaradas com `var` possuem escopo de função, toda vez que atribuímos um novo valor a variável `i` , na verdade estamos atualizando o valor da mesma referência. Tanto que se atribuímos um valor para `i` fora do loop, ao iterar as funções do `arrayLet` , temos esse valor impresso no console várias vezes:

```
const arrayVar = [];
for(var i = 0; i < 5; i++) {
    arrayVar.push(function () {
          console.log(i);
    });
}

i = 10; // atribuindo um novo valor
```

```
arrayVar.forEach(function (funcao) {
   funcao(); // 10 10 10 10 10
});
```

O mesmo não acontece com o `let`, porque, dentro do escopo de bloco, ele sempre cria uma nova variável `i`.

Legibilidade

É bastante comum em aplicações JavaScript trechos como este:

```
var API_KEY = 'xxxxxxxxxxxx';
```

É bem comum usarmos letras maiúsculas para nomear variáveis que são constantes. Isso é um ato normal entre os desenvolvedores, quase que uma boa prática. Entretanto, mesmo com essa indicação, nada nos impede de atribuir outro valor a esta variável. Com o `const`, não temos mais este problema, já que:

- O próprio desenvolvedor, ao ver a declaração com `const`, já sabe que se trata de uma constante, logo, ele não deve mudar este valor;
- Mesmo que o desenvolvedor seja mal-intencionado, o próprio ES6 não vai permitir que sejam atribuídos outros valores à constante.

A diferenciação entre constantes ou não também torna o entendimento do código mais claro. Em aplicações grandes e complexas, o código JavaScript pode ser bem extenso e cheio de variáveis dentro de vários contextos diferentes, o que torna a interpretação do código (e possivelmente sua execução e debug) bastante complicada.

Só com isso, o ES6 já torna nossa vida um pouco mais fácil.

Hoisting

No JavaScript, a declaração das nossas funções e variáveis possuem o comportamento de *Hoisting*. Este nome é dado ao comportamento de mover declarações para o topo de um escopo (global ou não).

Em outras palavras, isso significa que é possível usar uma variável ou função antes mesmo de declará-las no código. Como neste exemplo, no qual chamamos a função `imprimirNome` antes de realmente declarar a função:

```
imprimirNome('Tabata');

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

// Tabata
```

Este código funciona porque, antes de ser executado, a declaração da função `imprimirNome` é movida para o topo do escopo. Então, na verdade, o que é executado é:

```
function imprimirNome(nome) {
    console.log(nome);
}

imprimirNome('Tabata'); // Tabata
```

No ES6, o "hoisting" do `let` e `const` são diferentes de variáveis e funções. Quando uma variável é declarada usando `let` ou `const`, ela possui o que é chamada de *Temporal Dead Zone* (TDZ), nome que descreve o comportamento de que, no seu escopo, ela é inacessível até que a execução alcance sua declaração. Este comportamento é visível no trecho de código a seguir, em que:

- Criamos uma variável `valor` fora do escopo de bloco do `if`;
- Tentamos acessar a variável `valor`;
- Definimos a variável `valor`;

- Atribuímos o valor `1` à variável `valor`;
- Acessamos a variável `valor`;
- Acessamos a variável fora do escopo de bloco do `if`.

```
let valor = 0;

if (true) {
    // novo escopo, o TDZ do 'valor' começa

    // Ao tentar acessar a variável, tomamos ReferenceError,
    // pois neste momento ela é uma variável
    // que não foi inicializada
    console.log(valor); // ReferenceError

    let valor; // TDZ termina e 'valor' é definida com 'undefined'
    console.log(valor); // undefined

    valor = 1;
    console.log(valor); // 1
}

console.log(valor); // 0
```

Este comportamento evita que tenhamos resultados estranhos com os valores das variáveis, que era uma das grandes dores de cabeça ao usar JavaScript.

Últimas considerações

Por fim, antes de sair usando o `let` e o `const`, é importante ter em mente que:

- O `let` e o `const` sempre são escopados aos blocos mais próximos.
- O valor de uma variável `const` deve obrigatoriamente ser definido na sua declaração. Caso contrário, dará erro!
- O uso do `var` se tornou proibido, mas deve ser evitado.

CAPÍTULO 9

MANIPULAÇÃO DE TEXTOS COM TEMPLATE STRINGS

Uma das tarefas mais árduas no desenvolvimento de um sistema, seja ele desktop, mobile ou web, é lidar com palavras, frases e textos. Lidamos com eles em uma infinidade de atividades quando estamos fazendo uma aplicação:

- Mensagens de texto exibidas para os usuários;
- Construção de logs e auditoria;
- Mecanismos de buscas;
- Tratamento de textos (leitura/escrita);
- Ordenação;
- Entre outros.

Felizmente, para nós, grande parte das linguagens de programação de alto nível já possuem a estrutura de `String` , que nada mais é do que uma abstração de alto nível de uma cadeia de caracteres (em linguagens de baixo nível como C, por exemplo, o tratamento é ainda mais complicado, pois não há nativamente o tipo `String`).

A estrutura de `String` nos oferece muitas regalias, já que com ela conseguimos facilmente:

- Ler e escrever palavras, frases e textos;

- Descobrir seu tamanho;
- Criar substrings;
- Fazer diferenciação e ordenação;
- Buscar elementos específicos (com expressões regulares);
- Eliminar e transformar;
- E por aí vai.

Mas como vida de programador não é nada fácil, lidar com strings ainda tem vários probleminhas. Um exemplo bem clássico é a construção de mensagens para o usuário. Normalmente, temos de recuperar vários elementos pertinentes ao contexto em que o usuário está (tela, operação, login, data etc.), juntar tudo em um único texto e exibir na tela. Potencialmente, você já deve ter feito algo parecido com isto:

```
const login = 'ecmascript';
const dia = '13 de Setembro';
const ano = 2016;

const mensagem = "Olá, " + login + "!\nHoje é: " + dia + " de " +
ano;
console.log(mensagem);

// saída:
// Olá, ecmascript!
// Hoje é: 13 de Setembro de 2016
```

Só neste exemplo já temos dificuldades recorrentes:

- Somos obrigados a concatenar várias strings para formar a frase e inserir o valor das variáveis dentro dela;
- A variável `mensagem` , além de ficar grande, é pouco legível;
- Precisamos lidar com quebras de linha com caracteres especiais (`\n` , `\r`) para manter a formatação.

Para aliviar estas dores de cabeça, o ES6 trouxe o conceito de

template strings. Há dois tipos:

- Template strings simples;
- Template strings marcardos (tags).

Esses templates são estruturas que nos permite formar expressões com strings usando funcionalidades como interporlação e multilinhas. Ambos os tipos são features emprestadas da linguagem de programação E (http://erights.org/elang/index.html), criada em 1997 por Mark S. Miller e Dan Bornstein na Eletric Communites. Nesta linguagem, os templates strings são chamados de *quasi literals* (http://erights.org/elang/grammar/quasi-overview.html).

> **NÃO CONFUNDA!**
>
> Apesar de usarmos o nome "template", essa estrutura não tem nada a ver com o sentido de layout com que estamos acostumados, em que há um formato preestabelecido com campos que são preenchidos por valores arbitrários.

Estudaremos detalhadamente estes tipos.

9.1 TEMPLATE STRINGS SIMPLES

Qualquer desenvolvedor que já tenha trabalhado minimamente com JavaScript (ou até mesmo outras linguagens de programação) já deve ter feito um código semelhante a este:

```
var nome = 'Diego';
console.log('Bem-vindo, ' + nome); // Bem-vindo, Diego
```

Declaramos uma variável `nome` que contém um valor que é

exibido no console junto à mensagem `Bem-vindo`. Para fazer isso, tivemos de concatenar a mensagem com o nome utilizando o operador `+` (mais) e deixar um espaço proposital para que as palavras não ficassem grudadas.

Com o ES6, podemos concatenar strings de um jeito mais eficiente. Quer ver? É assim que fica o código anterior com ES6:

```
const nome = 'Diego';
console.log(`Bem-vindo, ${nome}`);
```

O primeiro item importante que precisamos reparar neste código é que não estamos mais usando aspas simples (`'`) na `String`, mas sim a crase (`). Usando esta nova sintaxe, a interpretação da `String` fica um pouco diferente. Com ela, podemos injetar o valor das variáveis diretamente dentro da `String`, sem a necessidade de concatenar. Basta inserir o valor da variável dentro do `${ }` (cifrão e chaves). Chamamos esta técnica de interpolação de strings.

Além de interpretar o valor das variáveis, também é possível inserir expressões dentro delas. Por exemplo, nesta soma de dois números:

```
const n1 = 1, n2 = 2;
console.log(`${n1} + ${n2} = ${n1 + n2}`);
// 1 + 2 = 3
```

Repare que usamos o valor das variáveis do lado esquerdo da operação (operadores), e do lado direito usamos o resultado da expressão `n1 + n2` como resultado. A expressão é interpretada e o resultado dela é injetado na `String`.

Outro ponto bacana sobre template strings é que a formatação é mantida. No primeiro trecho, deixamos naturalmente um espaço entre a frase `Bem-vindo,` e o resultado de `${nome}`, o que seria o equivalente a concatenar um espaço manualmente, como fizemos

no código ES5. A formatação da `String` fica intacta. Isso fica mais evidente no exemplo a seguir.

Como você escreveria o código dentro da função `console.log` para o que resultado seja este?

```
exibindo
uma
palavra
por
linha
```

Usando o ES5, temos de gerenciar todas as quebras de linha:

```
console.log("exibindo\numa\npalavra\npor\nlinha");
```

Agora comparemos com o mesmo comportamento usando o ES6:

```
console.log(`
exibindo
uma
palavra
por
linha
`);
```

Tanto o espaçamento entre as palavras quanto as quebras de linhas são mantidas, sem a necessidade de utilizar caracteres especiais (como `\n`, `\r`). Além da liberdade de poder escrever uma única `String` em múltiplas linhas.

Um excelente exemplo real de utilização para isso é a construção dinâmica de um nó HTML em uma variável que será inserido na DOM. Vamos imaginar que, em um cadastro de um site de compras, após o usuário preencher seus dados básicos — para manter simples, levaremos em conta só: nome, idade e endereço — surge um modal com os dados que ele acabou de colocar, para confirmar. O conteúdo do modal é gerado dinamicamente pelo JavaScript.

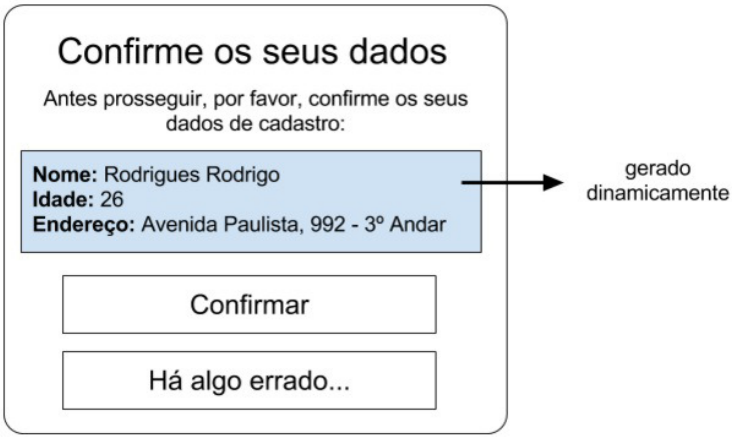


Figura 9.1: Representação gráfica do modal

Desconsiderando estilos CSS que seriam usados para modelar visualmente o conteúdo dentro do modal, o código JavaScript ES6 que monta este conteúdo do modal ficaria assim:

```
const div =
`<div>
  <p><b>Nome:</b> ${nome}</p>
  <p><b>Idade:</b> ${idade}</p>
  <p><b>Endereço:</b> ${endereco}</p>
</div>`;

// ... append do conteúdo no modal
```

Dentro de uma única variável, conseguimos estruturar um HTML com quebra de linhas, aninhamento e com interpolação. Agora para efeito de comparação, veja o código com JavaScript ES5 para ter o mesmo efeito:

```
var div = '<div>';
div += '<p><b>Nome:</b> ' + nome + '</p>';
div += '<p><b>Idade:</b> ' + idade + '</p>';
div += '<p><b>Endereço:</b> ' + endereco + '</p>';
div += '</div>';

// ... append do conteúdo no modal
```

9.2 TEMPLATE STRINGS MARCADO

Para este tópico, vamos utilizar a solução de um problema para explicar o funcionamento deste tipo de template strings. Imagine que temos um sistema onde exibimos ao usuário uma saudação junto ao horário atual em horas, algo como: "Bom dia, são 11 horas". Pensando no nosso código e aplicando o que aprendemos até aqui, podemos fazer:

```
const horas = new Date().getHours();
const mensagem = `Bom dia, são ${horas} horas`;
console.log(mensagem); // ex: Bom dia, são 12 horas
```

Aqui conseguimos fazer uso do template literal ao interpolar o valor das horas dentro da nossa `String` de mensagem. Muito bem. O problema, no entanto, é que a saudação tem de mudar de acordo com a seguinte regra:

Horários	Saudação esperada
6h - 12h	Bom dia
12h - 18h	Boa tarde
18h - 6h	Boa noite

Como podemos fazer isso? É ai que entra a mágica dos templates literais marcados. Começaremos marcando nossa mensagem com uma tag chamada `defineMensagem`. Repare que vamos colocar esta tag antes da crase e sem usar os parênteses:

```
const mensagem = defineMensagem`Bom dia, são ${horas} horas`;
```

Agora que marcamos nosso template, podemos usar esta tag para definir uma função que recebe dois parâmetros:

- Uma lista de strings;
- Cada uma das expressões do template.

Com essa função, temos o poder de dividir o template literal da

maneira que queremos. Vamos definir a função `defineMensagem` e colocar alguns logs no console para ter uma melhor ideia do que está acontecendo.

```
function defineMensagem(strings, ...values){
  console.log(strings);
  console.log(values);
}
```

Ao executar este código, temos como resultado:

```
["Bom dia, são ", " horas", raw: Array[2]]
[20]
```

Tendo acesso às strings e aos valores, fica fácil definir uma lógica que retorne a mensagem correta. Primeiro, tornamos a parte da saudação da mensagem em uma variável:

```
const mensagem = defineMensagem`${""}${horas} horas`;
```

Agora trabalhamos na lógica da função `defineMensagem` para atender ao requisito:

```
function defineMensagem(strings, ...values){
  const hora = values[1];
  if(hora >= 6 && hora <= 12) {
    values[0] = "Bom dia";
  } else if (hora > 12 && hora <= 18) {
    values[0] = "Boa tarde";
  } else {
    values[0] = "Boa noite";
  }

  values[0] = `${values[0]}, são`;
  return `${values[0]}${strings[0]}${hora}${strings[2]}`
}
```

Agora podemos testar nosso código com vários horários diferentes:

Horas	Resultado
10	Bom dia, são 10 horas
14	Boa tarde, são 14 horas

21	Boa noite, são 21 horas

Missão cumprida com sucesso. Bem bacana, não?

Qual a diferença, no final das contas?

É provável que você esteja se perguntando: qual a diferença em usar template strings marcado e usar uma função? Eu não poderia escrever:

```
const mensagem = defineMensagem(`Bom dia, são ${horas} horas`);
```

E daria na mesma?

Quase, mas na verdade não. Para ter os mesmos efeitos que termos com os templates, precisaríamos escrever uma função semelhante a esta:

```
function defineMensagem(['','horas'], horas){
    // corpo da função
}
```

O primeiro parâmetro seria as strings da nossa mensagem, e o segundo, a variável. Somente desta maneira conseguiríamos ter o mesmo efeito.

CAPÍTULO 10

ARROW FUNCTIONS

Uma função nada mais é do que um conjunto de instruções que executa uma tarefa específica, sendo que essa tarefa pode retornar um valor ou não. Para utilizá-las, precisamos defini-las em algum lugar e de algum modo no nosso código. No JavaScript, nós podemos declarar funções basicamente de três maneiras:

- Por meio de declaração de função (*function declaration*);
- Utilizando expressão de função (*function expression*);
- Invocando o construtor de `Function`.

A declaração de função e a expressão de função — as abordagens mais utilizadas para declarar funções — são muito similares e possuem quase a mesma sintaxe. A principal diferença é que, na declaração de uma função, é dado um nome para ela, e esta não necessita estar atribuída a uma variável, o que não acontece nas expressões, em que eles podem ser omitidos para criar as famosas funções anônimas.

Para ambas, usamos a palavra reservada `function` seguida da lista de seus parâmetros entre `()` (parênteses) e o seu corpo definido entre `{}` (chaves).

```
// declaração de função
function desligar() {
    // corpo da função
}
```

```
// expressão de função
var funcaoDesligar = function() {
    // corpo da função
}
```

Também podemos utilizar o construtor de `Function`, apesar deste ser um método pouco usado. O construtor aceita vários parâmetros, sendo que os primeiros são os argumentos da função e o último o corpo dela:

```
var funcaoDesligar = new Function('// corpo da função');
var funcaoDesligarComParametros = new Function(arg1,arg2,'// corpo
 da função');
```

Com o ES6, nós ganhamos mais uma maneira de declarar funções: através das arrow functions (em tradução livre, seria algo como "funções seta"). Esta estrutura é uma notação simplificada criada para facilitar a implementação de funções por expressão no JavaScript. A definição de uma arrow function é bem simples e segue esta ordem:

- Parâmetros dentro de parênteses (. . .);
- Fat arrow (=>);
- Corpo da função entre chaves (`{...}`).

Veja a sintaxe:

```
(param1, param2 …, param n) => {
    // corpo da função
}
```

Esta nova notação possui duas vantagens em relação a antiga maneira de declararmos expressões de função:

- São menos verbosas;
- O contexto de execução é diferente.

Vamos ver em detalhes cada uma destas vantagens.

10.1 SÃO MENOS VERBOSAS

Arrow functions nos permite escrever funções menos verbosas. Vamos mostrar partindo de um código ES5 e o refatorando passo a passo até o ideal em ES6. Considere a seguinte declaração de uma função anônima atribuída à variável `boasVindas`. Ela recebe um parâmetro chamado `nome` e o utiliza para retornar uma saudação:

```
var boasVindas = function(nome) {
    return "Seja bem-vindo, " + nome;
}

boasVindas("Luiz"); // Seja bem-vindo, Luiz
```

Esse código, apesar de já ser relativamente simples, pode ficar ainda mais enxuto usando arrow functions. Vamos começar a simplificá-lo removendo a palavra-chave `function` e adicionando a fat arrow (`=>`) no lugar:

```
var boasVindas = (nome) => {
    return "Seja bem-vindo, " + nome;
}

boasVindas("Luiz"); // Seja bem-vindo, Luiz
```

POR QUE O NOME FAT ARROW?

Fat arrow é uma contrapartida a thin arrow (`->`), utilizada na linguagem CoffeeScript.

Como só temos um parâmetro na nossa função, a sintaxe nos permite eliminar os parênteses:

```
var boasVindas = nome => {
    return "Seja bem-vindo, " + nome;
}

boasVindas("Luiz"); // Seja bem-vindo, Luiz
```

Como também só temos uma linha dentro do corpo da função, podemos eliminar a palavra-chave `return` e as chaves:

```
var boasVindas = nome => "Seja bem-vindo, " + nome;
boasVindas("Luiz"); // Seja bem-vindo, Luiz
```

Por fim, podemos dar um toque de excelência aplicando o que vimos no capítulo anterior e trocando o `var` por `const`:

```
const boasVindas = nome => `Seja bem-vindo, ${nome}`;
boasVindas("Luiz"); // Seja bem-vindo, Luiz
```

Perceba que continuamos com o mesmo comportamento, mas agora temos um código bem melhor.

10.2 O CONTEXTO DE EXECUÇÃO É DIFERENTE

Sempre que executamos uma função no JavaScript, ela é associada a um contexto de execução. Esse contexto possui uma propriedade denominada `ThisBinding`, que pode ser acessada a qualquer momento através da palavra reservada `this`. O valor do `this`, que chamamos de contexto da função, é constante e existe enquanto este contexto de execução existir.

Na maior parte dos casos, o valor do `this` é determinado pela forma como invocamos a função. Ele não pode ser assinado durante a execução, e isso pode ser diferente a cada vez que a função é chamada.

No navegador — também conhecido como contexto global —, o `this` referencia o objeto global `window`:

```
console.log(this); // Window {...}
```

Toda função também declarada no escopo global possui o objeto `window` como valor do `this`:

```
function imprimeMeuContextoDeExecucao() {
    console.log(this);
}

imprimeMeuContextoDeExecucao();
// Window {...}
```

Quando uma função representa um método de um objeto, o valor do `this` passa a ser o próprio objeto referenciado:

```
var objeto = {
  meuContexto: function () {
    console.log(this);
  }
};

objeto.meuContexto(); // { meuContexto: [Function: meuContexto] }
```

A situação começa a ficar confusa quando utilizamos o `this` dentro de uma função de `callback` e acabamos confundindo o seu valor. O `this` dentro do `callback` guarda o valor do objeto pai da função `callback` e não da função que recebe o `callback`. Confuso? Vejamos isso na prática.

Considere que temos um objeto chamado `equipe` e que nele temos duas propriedades: `nome` e `membros`. O nome é uma `String` que representa o nome da equipe, e o `membros` é um `Array` de nomes dos integrantes da equipe. Este objeto também tem um método `membrosDaEquipe` que, quando invocado, imprime no console o nome dos integrantes da equipe.

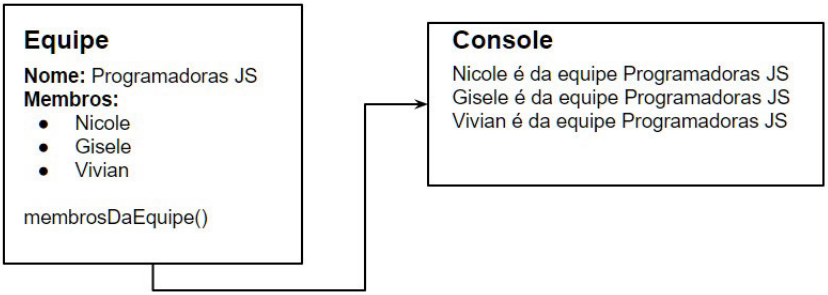


Figura 10.1: Representação do objeto equipe

Para tornar nosso exemplo mais divertido, vamos fazer uma homenagem ao antigo anime Dragon Ball Z, do Akira Toriyama, e criar uma equipe de guerreiros. Essa equipe de guerreiros possui a alcunha de "Guerreiros Z" e, neste exemplo, será formada por apenas três: Goku, Kuririn e Vegeta. Implementando o modelo, temos:

```
const equipe = {
    nome: 'Guerreiros Z',
    membros: ['Goku', 'Kuririn', 'Vegeta'],
    membrosDaEquipe: function() {
        // corpo da função
    }
}
```

Para exibir o nome dos membros, usaremos o contexto de execução para referenciar os membros e o nome da equipe. Para iterar sobre os membros, vamos usar a função `forEach` que aprendemos nos capítulos anteriores:

```
const equipe = {
    nome: 'Guerreiros Z',
    membros: ['Goku', 'Kuririn', 'Vegeta'],
    membrosDaEquipe: function() {
        this.membros.forEach(function(membro) {
            console.log(`${membro} é da equipe ${this.nome}`);
        });
    }
}
```

Na função `membrosDaEquipe`, esperamos que o `this` referencie o objeto `equipe`, de modo que para cada um dos membros da equipe, seja exibida a mensagem:

```
<membro> é da equipe Guerreiros Z
```

Entretanto, ao executar o código, para nossa surpresa, recebemos o seguinte erro:

```
Cannot read property 'nome' of undefined
```

Estranho, não é? Pois é. Esse tipo de situação é comum quando estamos desenvolvendo aplicações com JavaScript e não tomamos os devidos cuidados.

Este erro acontece porque, na nossa função anônima, o contexto de execução é diferente do contexto de execução do objeto. Como são diferentes, o `this` possui um outro valor diferente do que esperamos. Para resolver, podemos utilizar um velho truque do JavaScript: atribuir o valor do `this` em uma outra variável. Vamos chamá-la de `that`, e então invocá-la na função de `callback`:

```
const equipe = {
    nome: 'Guerreiros Z',
    membros: ['Goku', 'Kuririn', 'Vegeta'],
    membrosDaEquipe: function() {
        const that = this;
        this.membros.forEach(function(membro) {
            console.log(`${membro} é da equipe ${that.nome}`);
        })
    }
}
```

O que acabamos de fazer foi uma pequena malandragem para conseguir enxergar dentro do `callback` o contexto do pai. Agora, ao executarmos o `equipe.membrosDaEquipe`, temos a saída esperada:

```
Goku é da equipe Guerreiros Z
Kuririn é da equipe Guerreiros Z
Vegeta é da equipe Guerreiros Z
```

Para funcionar do jeito que queríamos, fomos obrigados a atribuir o valor do `this` em uma outra variável que chamamos de `that`. Fizemos isso pois, dentro do escopo da função anônima, não temos acesso ao contexto do `this` que contém o `nome`. Esta é uma situação recorrente em aplicações JavaScript. Além de ser muito confuso, as chances de causar grandes dores de cabeça e problemas é altíssima.

Para resolver o problema de contexto sem ser refém de truques, as arrow functions foram projetadas para conseguirmos capturar o `this` do seu contexto delimitador (chamamos isso de escopo léxico da função). Observe como só de trocar a implementação do exemplo para a sintaxe de arrow functions, nosso problema é resolvido:

```
const equipe = {
    nome: 'Guerreiros Z',
    membros: ['Goku', 'Kuririn', 'Vegeta'],
    membrosDaEquipe: function() {
        this.membros.forEach(membro => {
            console.log(`${membro} é da equipe ${this.nome}`);
        })
    }
}
```

Ao executar o `equipe.membrosDaEquipe` novamente, continuamos tendo a saída esperada:

```
Goku é da equipe Guerreiros Z
Kuririn é da equipe Guerreiros Z
Vegeta é da equipe Guerreiros Z
```

Como as arrow functions conseguem fazer a associação (bind) do `this` de forma automática, referenciar o `this` do contexto da execução delimitadora para o escopo da atual função deixou de ser um problema.

E o método bind?

Outra maneira de resolver o problema que tratamos no item anterior — e que você talvez já tenha ouvido falar — é pelo método `bind`. Este método, introduzido no ES5, ao ser invocado, cria uma nova função com o mesmo corpo e escopo da função com que foi associado, mas com o `this` da função original.

Para demonstrar, considere a função `mostrarPropriedadeDoContexto` que recebe um parâmetro chamado `nomePropriedade` e o recupera do seu contexto de execução para imprimir no console:

```
function mostrarPropriedadeDoContexto(nomePropriedade) {
  console.log(this[nomePropriedade]);
}
```

Vimos que, fora de um objeto, a função utiliza o contexto de execução global, ou seja, o `this` será igual ao objeto `window` do navegador. Ao executar este método procurando pela propriedade `location`, por exemplo, receberemos uma resposta como esta:

```
// equivalente a window.location
{
    replace: [Function],
    assign: [Function],
    hash: '',
    search: '',
    pathname: 'blank',
    port: '',
    hostname: '',
    host: '',
    protocol: 'about:',
    origin: 'null',
    href: 'about:blank',
    ancestorOrigins: {
        '0': 'https://replit.org',
        '1': 'https://replit.org',
        '2': 'https://repl.it'
    },
    reload: [Function: reload]
}
```

Agora, se usamos o método `bind` para atribuir um novo valor

de `location` no contexto da função, podemos ter a resposta que quisermos:

```
var mockLocation = {
    location : 'fake-location'
}

var funcao = mostrarPropriedadeDoContexto.bind(mockLocation);
```

Ao executar o `funcao('location')`, notaremos que o contexto de execução foi "sobrescrito", de modo que agora não está mais referenciando o objeto `window`, mas sim o objeto `mockLocation`:

```
fake-location
```

Essa seria a mesma situação para o caso da `equipe`. Para conseguir utilizar o `this.nome` corretamente, basta usar o `bind` na função anônima de iteração, pois deste modo estamos passando o contexto de execução que conhece o valor de `nome` para a função:

```
const equipe = {
    nome: 'Guerreiros Z',
    membros: ['Goku', 'Kuririn', 'Vegeta'],
    membrosDaEquipe: function() {
        this.membros.forEach(function(membro) {
            console.log(`${membro} é da equipe ${this.nome}`);
        }.bind(this));
    }
}
```

Ao executar `membrosDaEquipe`, temos o mesmo resultado de antes:

```
Goku é da equipe Guerreiros Z
Kuririn é da equipe Guerreiros Z
Vegeta é da equipe Guerreiros Z
```

Considerações finais

Em resumo, as arrow functions nos permitem criar funções mais enxutas e com o poder de acessar o `this` do seu contexto

delimitador, sem a necessidade de fazer truques, como `var that = this` ou utilizar o método `bind`.

CAPÍTULO 11

MELHORIAS EM OBJETOS LITERAIS

Apesar de não possuir classes (até o ES6) e nem interfaces, o JavaScript é uma linguagem de programação orientada a objetos (baseada em prototipagem). Para o JavaScript, um objeto é uma coleção de propriedades e cada propriedade é uma associação entre chave/valor — como vimos no capítulo de Mapas. O valor pode ser primitivo, assim como uma função, que é então considerada um método do objeto. Além dos objetos que já são preestabelecidos no navegador, como o objeto `window`, é possível definir nossos próprios objetos.

Existem duas maneiras de se criar objetos no JavaScript: por meio de funções construtoras, ou por objetos literais. Os construtores são funções que, quando invocadas com a palavra reservada `new`, criam uma nova instância de um objeto, como neste segmento:

```
function Livro(titulo) {
    this.titulo = titulo;
}

var livro = new Livro('Entendendo ES6');
console.log(livro.titulo); // Entendendo ES6
```

Quando executamos a função `Livro` acompanhada do `new`, quatro coisas acontecem por debaixo dos panos:

- Um novo objeto literal é criado;
- O construtor do objeto `livro` é definido como Livro, assim como o seu tipo (que pode ser verificado com o `instanceof`);
- O protótipo do objeto `livro` é definido como `Livro.prototype`;
- É criado um novo contexto de execução para o objeto.

A diferença crucial das funções construtoras em relação aos objetos literais é que estes são estáticos e únicos. Isso significa que mesmo que ele seja armazenado em diferentes variáveis, todas apontarão para a mesma referência. Repare:

```
var livro = {
    titulo: 'Anjos e Demônios'
}

console.log(livro.titulo); // Anjos e Demônios

var outroLivro = livro;
livro.titulo = 'O Código da Vinci';

console.log(outroLivro.titulo); // O Código da Vinci
console.log(livro.titulo); // O Código da Vinci
```

Ao atribuir `livro` a variável `outroLivro`, ambas apontam para a mesma referência. Quando alteramos uma das propriedades, ela se reflete nas duas variáveis, pois se tratam do mesmo objeto. Mas ainda que sejam estáticos e únicos, eles são altamente flexíveis. Podemos acessar e modificar suas propriedades e adicionar novas utilizando a sintaxe de `.` (ponto) ou `[]` (colchetes).

```
var livro = {
    titulo: 'Anjos e Demônios'
}

livro.autor = 'Dan Brown';
livro['mostrarLivro'] = function () {
    console.log(this.titulo + ', ' + this.autor);
}
```

```
livro.mostrarLivro(); // Anjos e Demônios, Dan Brown
```

Com o ES6, ganhamos várias melhorias para nos ajudar a manipular objetos literais. Veremos todas ao decorrer do capítulo.

11.1 DECLARAÇÃO DE PROPRIEDADES

A primeira melhoria diz respeito a declaração de propriedades dentro de um objeto literal. Tome como nota este trecho de código em ES5:

```
var nome = 'Maria';
var sobrenome = 'Madalena';

var pessoa = {
    nome: nome,
    sobrenome: sobrenome
}
```

Criamos duas variáveis (`nome` e `sobrenome`) e atribuímos seus valores ao objeto `pessoa` , usando o próprio nome da variável como nome da propriedade. Com o ES6, quando temos propriedades e variáveis com nomes iguais, podemos declarar as propriedades passando somente o nome uma vez:

```
const nome = 'Maria';
const sobrenome = 'Madalena';
const pessoa = {nome, sobrenome};
```

O próprio interpretador já associa o nome da propriedade com a variável de mesmo nome dentro do seu escopo. Se verificarmos os valores das propriedades `nome` e `sobrenome` nos dois casos, nos certificamos de que o valor é exatamente o mesmo:

```
console.log(pessoa.nome); // Maria
console.log(pessoa.sobrenome); // Madalena
```

E o mesmo princípio é válido para declaração de funções:

```
const nome = 'Maria';
const sobrenome = 'Madalena';
```

```
const seApresentar = function() {
    console.log(`Olá! Sou a ${this.nome} ${this.sobrenome}`);
}

const pessoa = {nome, sobrenome, seApresentar};
pessoa.seApresentar(); // Olá! Sou a Maria Madalena!
```

Na verdade, no caso das funções, há outra melhoria bem bacana. No exemplo anterior, podemos remover o sinal de igualdade, a palavra reservada `function` e passar a declaração do método diretamente para dentro do objeto `pessoa`:

```
const nome = 'Maria';
const sobrenome = 'Madalena';

const pessoa = {
    nome,
    sobrenome,
    seApresentar(){
        console.log(`Olá! Sou a ${this.nome} ${this.sobrenome}`);
    }
};

pessoa.seApresentar(); // Olá! Sou a Maria Madalena
```

São melhorias triviais, mas nos ajudam a manter o código limpo.

11.2 ÍNDICES DE PROPRIEDADES COMPUTADAS

Além de permitir a declaração de objetos literais com uma sintaxe abreviada, o ES6 também nos permite ter índices de propriedades computadas em uma definição de objeto literal. Isso significa que podemos passar expressões no qual o resultado será equivalente ao nome que será relacionado à chave. Em termos práticos, quer dizer que podemos fazer algo assim:

```
const nomeMetodo = 'invocar';
const objeto = {
    [nomeMetodo](){
        console.log('executou método');
    }
```

```
}

objeto[nomeMetodo](); // executou método
```

Passamos o nome da propriedade do `objeto` como sendo equivalente ao valor atribuído a variável `nomeMetodo`. Mas temos como ir além. Podemos fazer qualquer tipo de concatenação entre strings ou avaliação dentro os `[]` (colchetes) que o resultado da expressão será computado como a chave daquela propriedade. Veja o exemplo a seguir, no qual fazemos com que o `objeto` tenha uma propriedade chamada `mostrarNome` por meio da concatenação de outras duas variáveis:

```
const nomeFuncao = 'mostrar';
const propriedade = 'Nome';

const objeto = {
    Nome : 'Objeto',
    [`${nomeFuncao}${propriedade}`](){
        console.log(this[propriedade]);
    }
}

objeto.mostrarNome(); // Objeto
```

Impressionante, não é mesmo? E o mesmo é válido para propriedades que não são métodos:

```
const apelido = "apelido";
const pessoa = {
    nome: 'José'
    [apelido]: 'Zé'
}

pessoa[apelido]; // Zézinho
```

11.3 OBJETOS LITERAIS X JSON

O *JavaScript Object Notation* (JSON) é um formato leve, criado como subconjunto da notação de objetos literais do JavaScript, para troca de dados. Originalmente criado por Douglas Crockford, o

formato que é reconhecido devido a sua simplicidade e flexibilidade, não é usado somente no JavaScript, mas na maior parte das linguagens de programação voltadas para web.

Grande parte das empresas hoje que oferecem APIs do tipo *Representational State Transfer* (REST) utilizam o formato para comunicação, em que temos a requisição e a resposta em JSON (*application/json*). Muitos dos aplicativos e sites que usamos em nossos navegadores e smartphones se comunicam com serviços terceiros através de JSON para realizar suas tarefas. Alguns exemplos bem famosos de empresas/serviços que oferecem APIs em que sua comunicação acontece por JSON:

- Redes sociais (Facebook, Twitter, Google+)
- Plataformas de pagamentos online (PayPal, Cielo, PagSeguro)
- Serviços de localização (Google Maps, Foursquare)
- Plataformas de comércio eletrônico (Mercado livre, Amazon)
- Serviços de comparação de preços (Buscapé, Indix)

Vamos ver com um exemplo prático como lidar com JSON e como as melhorias do ES6 neste aspecto nos ajudam. Como dissemos, o JSON nada mais é que um subconjunto da notação de objetos literais que vimos durante o decorrer do capítulo, com a simples diferença de que:

- Todas as chaves precisam estar entre aspas;
- Strings precisam estar sempre entre aspas duplas;
- Os valores possuem limitações (por exemplo, não podem ser funções).

O JSON pode ser usado tanto na resposta (*response*) quando na solicitação (*request*) de uma API. Imagine que estamos construindo um sistema de registro de pacientes para uma grande rede de

hospitais. Como essa rede é bem grande e já está no mercado há muito tempo, ela já possui uma série de sistemas atuando em sua infraestrutura. Isso significa que a nossa aplicação vai ter de se integrar por meio dos outros sistemas através de uma API que eles oferecem.

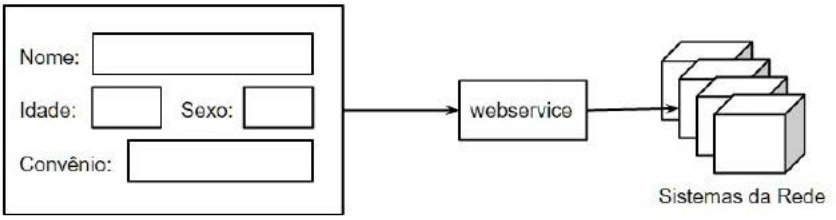


Figura 11.1: Protótipo da aplicação de registro de pacientes

Primeiro, precisamos obter os dados que serão preenchidos na tela pelo usuário. Vamos imaginar que no front-end, queremos obter os dados dos `inputs` com id: `nome`, `idade`, `sexo` e `convenio`. Para isso, criamos um método chamado `obterDadosDaTela` que obtém os dados e retorna um objeto literal:

```
function obterDadosDaTela() {
    const nome = document.getElementById('nome').value;
    const idade = document.getElementById('idade').value;
    const sexo = document.getElementById('sexo').value;
    const convenio = document.getElementById('convenio').value;

    return {nome,idade,sexo,convenio};
}
```

Lendo a documentação da API dos sistemas dos hospitais, descobrimos que precisamos enviar uma solicitação HTTP para `https://sistemasaude.com.br/api/cadastros/paciente`, para fazer o registro do paciente. Essa solicitação tem alguns requisitos:

- Precisa ser `POST`;
- O `Content-Type` precisa ser especificado como

`application/json` ;

- Os dados precisam ser enviados em um JSON.

Podemos usar o JQuery (https://jquery.com/) para nos ajudar a fazer isso. Ele possui o método `$.ajax` que realiza um AJAX e nos deixa especificar todos os requisitos solicitados.

```
const url = 'https://sistemasaude.com.br/api/cadastros/paciente';
const dados = obterDadosDaTela();

// transformamos nosso objeto em JSON
const dadosJson = JSON.stringfy(dados);

$.ajax({
    url        : url,
    dataType   : 'json',
    contentType: 'application/json; charset=UTF-8',
    data       : dadosJson,
    type       : 'POST',
    complete   : callback // etc
});
```

Pronto! Já conseguimos integrar os sistemas! Note que mesmo sendo um exemplo fictício e bem simples, ele representa uma situação bastante recorrente no desenvolvimento de aplicações web, que é o de integração entre sistemas/serviços.

Com a explosão do ecossistema Node.js e NPM, o JavaScript ganhou muita força e cada vez mais surgem serviços com APIs que trabalham com JSON. Saber como trabalhar com elas é estar um passo à frente.

Capítulo 12

PARÂMETROS PREDEFINIDOS EM FUNÇÕES

No JavaScript, quando declaramos e invocamos funções, elas possuem por definição seus parâmetros com o valor `undefined` como padrão. Na prática, isso significa que, se não indicamos explicitamente qual o valor que queremos que o parâmetro tenha na execução do método, o valor assumido sempre será `undefined`:

```
function minhaMaeMandouEuEscolherEsseDaqui(qualEuEscolhi) {
    console.log(qualEuEscolhi);
}

minhaMaeMandouEuEscolherEsseDaqui('terceiro'); // terceiro
minhaMaeMandouEuEscolherEsseDaqui(); // undefined
```

Temos o mesmo comportamento com métodos que possuem múltiplos parâmetros:

```
function imprimeNomeCompleto(nome, sobrenome, nomeDoMeio) {
    console.log(`${nome} ${nomeDoMeio} ${sobrenome}`);
}

imprimeNomeCompleto('Jorge', 'Reis'); // Jorge Reis undefined
imprimeNomeCompleto('Isabela', 'Joana Luiza', 'Jesus');
// Isabela Joana Luiza Jesus
```

Reparem que, quando não passamos o `nomeDoMeio`, nosso código imprime o valor da variável como `undefined`. Para qualquer parâmetro que a função esteja esperando que não seja

definida, é assumida como `undefined` na execução.

Nestes casos, para não termos valores indefinidos na execução das funções, o hábito dos desenvolvedores até então era testar se o valor de uma variável é `undefined`, para então lhe atribuir um valor predefinido, ou fazer o código passar por um fluxo diferente. Veja este exemplo:

```
function imprimeNomeCompleto(nome, sobrenome, nomeDoMeio) {
    if(nomeDoMeio === undefined) {
        console.log(`${nome} ${sobrenome}`);
    } else {
        console.log(`${nome} ${nomeDoMeio} ${sobrenome}`);
    }
}

imprimeNomeCompleto('Romulo', 'Scampini'); // Romulo Scampini
imprimeNomeCompleto('Leticia', 'Aparecida', 'de Souza');
// Leticia Aparecida de Souza
```

Se o `nomeDoMeio` não é definido na chamada do método, o controle do fluxo valida e imprime no console o nome de acordo com o que foi passado.

Com o ES6, não é preciso mais fazer este tipo de lógica, porque agora é possível atribuir valores predefinidos aos parâmetros de nossas funções, de modo de que eles assumem estes valores quando não são definidos na execução. Vamos ver como isso funciona.

12.1 ATRIBUINDO VALORES PADRÕES

Agora é possível definir valores padrões aos parâmetros na declaração das nossas funções. Pegando o exemplo anterior, em vez de fazer uma verificação para saber se o valor do `nomeDoMeio` era `undefined` dentro da implementação do método, podemos fazer isso:

```
function imprimeNomeCompleto(nome, sobrenome, nomeDoMeio = "") {
    console.log(`${nome} ${nomeDoMeio} ${sobrenome}`);
```

```
}

imprimeNomeCompleto('João', 'da Silva', 'Aparecido');
// João Aparecido da Silva
imprimeNomeCompleto('João', 'da Silva'); // João da Silva
```

Repare que, em sua declaração, atribuímos o valor `""` (espaço em branco) a variável `nomeDoMeio`. Desta maneira, se na invocação do método não for passado o valor do parâmetro, ele automaticamente assume este valor predefinido. Esta abordagem tem muitas vantagens, já que evita que façamos invocações deste tipo:

```
imprimeNomeCompleto('João', '', ''); // João
```

Também evita que façamos lógicas desnecessárias, como no caso do `if` ou no caso seguinte, pois não é necessário validar mais os valores:

```
function imprimeNomeCompleto(nome, sobrenome, nomeDoMeio) {
  let sobrenomeTratado = sobrenome || '';
  let nomeDoMeioTratado = nomeDoMeio || '';

  console.log(`${nome} ${nomeDoMeio} ${sobrenome}`);
}

imprimeNomeCompleto('João'); // João
```

Ou mesmo o que é pior: implementar duas funções que possuem praticamente o mesmo comportamento.

```
function imprimeNomeCompleto(nome) {
  console.log(nome);
}

function imprimeNomeCompleto(nome, sobrenome) {
  console.log(`${nome} ${sobrenome}`);
}

imprimeNomeCompleto('João'); // Seu nome é: João
imprimeNomeCompleto('João', 'Silva'); // Seu nome é: João Silva
```

Bem empolgante essa funcionalidade, né? Mas, antes de sair

usando, é necessário estar atento aos detalhes importantes. Sabê-los é o que vai fazer a diferença. Vamos explorá-los.

12.2 VALORES UNDEFINED

Precisamos ter uma atenção especial a variáveis com valor `undefined`. Uma vez que definimos uma função que contenha um parâmetro com valor predefinido, se a invocarmos e o parâmetro correspondente for equivalente a `undefined` na hora de sua execução, o valor predefinido que será considerado. Vamos entender melhor o que isso quer dizer vendo um pouco de código. Considere a função `multiplicaPor` declarada:

```
function multiplicaPor(valor, multiplicador = 2) {
  return valor * multiplicador;
}

const valor = multiplicaPor(2,2);
console.log(valor); // 4
```

Se chamarmos a mesma função com o segundo parâmetro como `undefined`, teremos o mesmo resultado:

```
function multiplicaPor(valor, multiplicador = 2) {
  return valor * multiplicador;
}

const valor = multiplicaPor(2, undefined);
console.log(valor); // 4
```

Para que o valor do argumento seja intencionalmente interpretado como um valor desconhecido e/ou que não existe, precisamos utilizar o `null` em vez do `undefined`:

```
function print(valor = '') {
    console.log(valor);
}

print(); // ''
print(null); // null
```

Precisamos ter muito cuidado ao trabalhar com variáveis que podem assumir o valor de `undefined`.

12.3 REFERENCIANDO OUTROS VALORES PADRÕES

Podemos definir o valor padrão de um parâmetro como sendo o valor padrão de outro parâmetro vizinho. Como neste método `calculaPotencia`:

```
function calculaPotencia(x = 2, y = x){
  console.log(Math.pow(x, y));
}

calculaPotencia(); // 4
calculaPotencia(2); // 4
calculaPotencia(2,2); // 4
```

Na primeira ocorrência, quando invocamos a função sem nenhum argumento, ele considerou o valor padrão que estabelecemos para a variável `x` e atribuiu o valor de `x` na variável `y`, o que resultou em dois ao quadrado. Na segunda ocorrência, especificamos o primeiro argumento, mas não o segundo. Novamente foi atribuído o valor das variáveis `x` a `y`, resultando mais uma vez em dois ao quadrado. Na terceira ocorrência, como passamos ambos parâmetros, o resultado foi o que já era esperado.

12.4 REFERENCIANDO VARIÁVEIS INTERNAS

Assim como podemos referenciar outros parâmetros que possuem valores padrões, também podemos referenciar outras variáveis que estão fora do escopo da declaração da função. Contudo, é necessário estar atentos aos seus escopos. Por exemplo, no trecho de código a seguir, tentamos atribuir o valor da variável `v` como valor padrão:

```
const v = 'valor 1';
function funcao(x = v) {
  const v = 'valor 2';
  console.log(x);
}

funcao(); // qual valor será mostrado?
```

Ao executar esta função, percebemos que o valor atribuído ao padrão é "valor 1" e não "valor 2". Isso aconteceu exatamente por causa do escopo de bloco que vimos no capítulo de `const` e `let`. Se removemos a primeira declaração da variável `v`, somos presenteados com um belo erro:

```
ReferenceError: v is not defined
```

12.5 UTILIZANDO FUNÇÕES COMO VALORES PADRÕES

Como você já deve suspeitar, podemos usar funções (anônimas ou não) como valores padrões de outras funções. Um bom exemplo de uso é a definição de um `callback` padrão para a chamada de uma função:

```
function facaAlgoComMeuNome(nome, callback = z => {
  console.log(z);
}) {
  callback(nome);
}

facaAlgoComMeuNome('Muriel'); // Muriel
```

12.6 TORNANDO PARÂMETROS OBRIGATÓRIOS

Agora vamos ver mais um caso no qual podemos aplicar estas melhorias na prática. Imagine que estamos construindo uma biblioteca JavaScript para construir componentes de telas, algo

como a biblioteca React do Facebook.

Nesta biblioteca, criamos o método `inserirNaTela(objeto)`, em que `objeto` é um parâmetro obrigatório, como descrevemos na documentação (ou vai me dizer que você não documenta o seu código?). Com esta novidade de parâmetros predefinidos em funções, podemos validar se a função que criamos realmente está sendo usada apropriadamente pelos desenvolvedores.

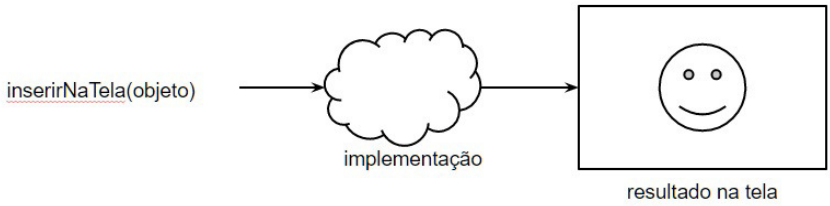


Figura 12.1: Representação do funcionamento da nossa biblioteca

Podemos impedir os desenvolvedores de utilizar o método da nossa API sem passar adequadamente o parâmetro `objeto`. Caso isso aconteça, devemos exibir um erro indicando quando é problema. Para tal, associamos um erro como um valor predefinido deste parâmetro:

```
function parametroObrigatorio(parametro) {
  throw new Error(`O parâmetro "${parametro}" é obrigatório!`);
}

function inserirNaTela(objeto = parametroObrigatorio('objeto')) {
  // lógica de implementação do método
}
```

Agora, ao executar o método sem passar nenhum parâmetro, recebemos o erro:

```
inserirNaTela();
// Error: O parâmetro "objeto" é obrigatório!
```

Esta é uma maneira interessante de impedir que os métodos

sejam executados sem os parâmetros obrigatórios.

Capítulo 13

PARÂMETROS INFINITOS COM OPERADOR REST

Em muitas situações nas nossas aplicações, queremos funções que saibam trabalhar com um número desconhecido de parâmetros. Um caso onde isso pode ser útil, por exemplo, é na construção de um método que gera uma query para ser executada no banco de dados.

Suponha que queremos um método que construa uma query SQL (*Structured Query Language*) de consulta simples — entenda consulta simples como uma consulta sem condições `where`. Nele passamos como parâmetros somente o nome da tabela e as colunas que queremos extrair.

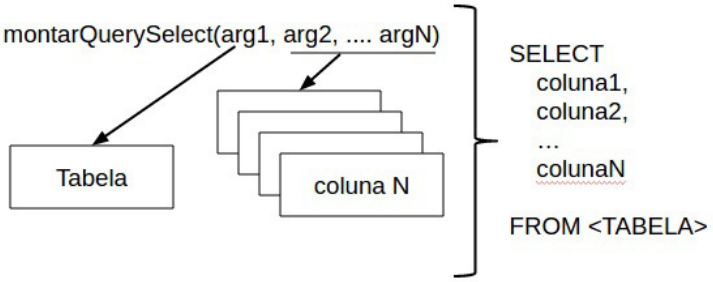


Figura 13.1: Lógica por trás do método que monta select

No ES5, podemos nos aproveitar do objeto `arguments` para fazer isso. Este objeto está disponível dentro de todas as funções

construídas no JavaScript. Ele contém um registro para cada argumento passado para a função no contexto de sua execução, sendo que o primeiro índice de registro começa no índice 0.

Sabendo disso, ficou fácil implementar nosso código. Vamos impor que somente o nome da tabela, o primeiro parâmetro, é obrigatório. Quando não forem passados outros parâmetros para as colunas, vamos utilizar o símbolo `*` (asterisco) — que representa uma busca que traz todas as colunas da tabela. Uma das implementações possíveis é esta:

```
function montaQuerySelect() {
    const tabela = arguments[0];
    const qtdArgs = arguments.length;
    let cols = '';
    if(qtdArgs > 1) {
        for(let index = 1; index < qtdArgs; index++) {
            cols += `${arguments[index]}, `;
        }
        cols = cols.substring(0, cols.length - 2);
    } else {
        cols = '*';
    }

    return `SELECT ${cols} from ${tabela}`;
}
```

Para atender aos dois casos, verificamos se a `qtdArgs` é maior que um. Se for, usamos o objeto `arguments` para recuperar todos os parâmetros passados e colocar em uma `String` que chamamos de `cols`. Ao final da iteração, usamos o `substring` somente para remover a última vírgula. No caso de não ser passado mais do que um argumento, é mais fácil, pois somente usamos o `*` (asterisco) no lugar.

Ao avaliar alguns casos, vemos que a implementação atende à nossa demanda:

```
const query1 = montaQuerySelect('tabela');
const query2 = montaQuerySelect('tabela', 'col1');
const query3 = montaQuerySelect('tabela', 'col1', 'col2');
```

```
// saída:
// query1 > SELECT * FROM tabela
// query2 > SELECT col1 FROM tabela
// query3 > SELECT col1, col2 FROM tabela
```

Com base no funcionamento do objeto `arguments`, foi criado o operador Rest para trazer novas melhorias no tratamento de parâmetros. Vamos ver detalhes do que exatamente o `arguments` nos oferece, como ele funciona e qual sua relação com o novo conceito Rest.

13.1 ENTENDA O QUE ARGUMENTS FAZ

O que o objeto `arguments` faz quando invocado dentro de uma função é nos retornar um objeto, com uma sintaxe muito semelhante à de um `Array` (mas que não é um! Isso é muito importante!), contendo uma referência para todos os argumentos que foram passados para o contexto de execução da função. Tome nota do seu uso no exemplo:

```
function logarTodosArgumentos() {
    for (let i = 0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}

logarTodosArgumentos(1,2,3); // 1, 2, 3
```

Dentro da nossa função, utilizamos o objeto `arguments` para recuperar a quantidade de argumentos passados com a propriedade `length` e seus respectivos valores através de índices. Um outro caso em que podemos fazer isso, por exemplo, é para facilmente criar uma função que recebe um número indefinido de parâmetros numéricos e devolve a soma de todos eles:

```
function somar(){
  let soma = 0;
  const qtd = arguments.length;
  for(let i = 0; i < qtd; i++) {
```

```
    soma += arguments[i];
  }

  return soma;
}

console.log(somar(1,2)); // 3
console.log(somar(1,2,3)); // 6
console.log(somar(1,2,3,4)); // 10
```

O `arguments` nos concede o poder de resgatar parâmetros da função mesmo que eles não tenham sido declarados na assinatura do método.

13.2 ARGUMENTS X OPERADOR REST

Agora que entendemos como o `argumets` funciona, vamos para as novidades. A sintaxe do recém-chegado operador Rest nos permite representar um número indefinido de argumentos em um `Array`. Se o último argumento nomeado de uma função estiver acompanhado de `...` (três pontos), ele vai se tornar um `Array` no qual os elementos são disponibilizados pelos argumentos atuais passados à função.

Vamos reescrever o método `somar` do exemplo anterior com o operador Rest para entender o que isso significa:

```
function somar(...valores){
  let soma = 0;
  const qtd = valores.length;
  for(let i = 0; i < qtd; i++) {
    soma += valores[i];
  }

  return soma;
}

console.log(somar(1,2)); // 3
console.log(somar(1,2,3)); // 6
console.log(somar(1,2,3,4)); // 10
```

Na assinatura do método, usamos `...valores` para representar uma quantidade indefinida de parâmetros passados para a função. Dentro da função, obtemos `valores` como um `Array`. A partir daí, bastou iterá-lo para obter os valores e somá-los.

Uma das principais diferenças entre o `arguments` e o operador Rest é que o primeiro não é realmente um objeto `Array`, enquanto o segundo é. Isso significa que, com os operadores Rest, podemos usar os métodos auxiliares de `Array` que vimos nos capítulos anteriores, tais como: `forEach`, `map`, `filter`, `find` etc. Além da possibilidade de poder utilizar o laço de iteração `for...of`.

Na nossa função `somar`, podemos melhorar sua implementação usando o método auxiliar `reduce` e a sintaxe de arrow function.

```
function somar(...valores){
  return valores.reduce((soma, valor) => {
    return soma + valor;
  }, 0);
}

console.log(somar(1,2)); // 3
console.log(somar(1,2,3)); // 6
console.log(somar(1,2,3,4)); // 10
```

Agora sim, nosso método ficou enxuto, objetivo e de acordo com as melhorias do ES6.

13.3 PARTICULARIDADES DO OPERADOR REST

Um ponto importante que precisamos ficar de olho diz respeito ao fato de que este operador só funciona no último argumento nomeado de uma função. Isso significa que não podemos utilizar mais de um operador Rest por função. Veja o seguinte caso, no qual

pretendemos separar números e letras:

```
function numerosELetras(...numeros, ...letras) {
  // corpo da função
}
```

Isto simplesmente não funciona. O operador sempre interpreta as últimas variáveis passadas na função para compactá-las em um único `Array`. Apesar de a separação parecer fazer sentido para a chamada a seguir, não funcionará como esperado.

```
function numerosELetras(1,2,3,'a','b','c') {
  // corpo da função
}
```

O correto para este caso é:

```
function numerosELetras(...numerosELetras) {
  // corpo da função
}
```

Sem a distinção de números e letras, podemos atribuir ambas no `Array` que é criado ao se utilizar o operador Rest.

13.4 PODEMOS UTILIZAR EM CONJUNTO COM PARÂMETROS "FIXOS"

Como o operador sempre interpretará unicamente os últimos argumentos passados na função, podemos definir variáveis que ficam de fora do operador. Vamos chamá-los de parâmetros fixos.

Para atestar o que isto significa, criamos uma função que faz a soma de inúmeros valores e, no final, multiplica essa soma por um outro valor. Esta função se chama `somaTudoEMultiplicaPor` e seu primeiro parâmetro `multiplicador` será fixo, enquanto os `...valores` podem variar.

```
function somaTudoEMultiplicaPor(multiplicador, ...valores){
  return valores.reduce((soma, valor) => {
    return soma + (valor * multiplicador);
```

```
  }, 0);
}

console.log(somaTudoEMultiplicaPor(2,1,2)); // 6
console.log(somaTudoEMultiplicaPor(2,6,7)); // 26
```

Note como o número atribuído ao `multiplicador` não faz parte do operador, porque não contém o prefixo `...`, somente seus argumentos vizinhos.

Por fim, podemos usar essa mesma estratégia para refatorar o nosso método `montaQuerySelect` que foi apresentado logo no início deste capítulo. Havíamos definido que o primeiro argumento sempre seria o nome da tabela correspondente. Fazendo uma associação com o exemplo que acabamos de ver, isto seria o nosso parâmetro fixo, enquanto as colunas podem variar. A refatoração fica assim:

```
function montaQuerySelect(tabela, ...cols) {
    let colsQuery= '';
    if(cols.length > 0) {
        colsQuery = cols.reduce((colsQuery, coluna) => {
            return colsQuery+= `${coluna}, `;
        }, '');
        colsQuery = colsQuery.substring(0, colsQuery.length -2);
    } else {
        colsQuery = '*';
    }

    return `SELECT ${colsQuery} FROM ${tabela}`;
}
```

Ao executar, confirmamos que os resultados continuam os mesmos:

```
const query1 = montaQuerySelect('tabela');
const query2 = montaQuerySelect('tabela', 'col1');
const query3 = montaQuerySelect('tabela', 'col1', 'col2');

// saída:
// query1 > SELECT * FROM tabela
// query2 > SELECT col1 FROM tabela
// query3 > SELECT col1, col2 FROM tabela
```

Capítulo 14

EXPANSÃO COM O OPERADOR SPREAD

Quando trabalhamos com funções no JavaScript, nós as utilizamos passando os parâmetros — sempre que aplicável — que desejamos que sejam considerados na sua execução. Um exemplo de método que recebe parâmetros e que já foi usado várias vezes ao longo do livro é a própria função `log` do console, que estamos usando para exibir valores no console do navegador.

Nós a invocamos sempre passando os valores que queremos que sejam exibidos. Normalmente passamos somente um único valor ou objeto, mas é possível passar vários:

```
console.log(1); // 1
console.log({}); // {}
console.log(1,2,3); // 1, 2, 3
```

Em muitos casos, os múltiplos argumentos que queremos passar para uma função estão contidos em uma lista. Então, para usá-los, temos de recuperá-los um a um, acessando os seus índices, e passá-los individualmente:

```
var argumentos = [1,2,3];
console.log(argumentos[0], argumentos[1], argumentos[2]);
// 1, 2, 3
```

No ES5, para evitar ficar fazendo isso e já passar de uma vez a lista inteira como argumento para a função, podemos usar a função `apply`, disponível em todas as funções do JavaScript. Este método

aceita dois parâmetros: o primeiro deles representa o contexto de execução (`this`) que será considerado; o segundo é um `Array` que representa os argumentos que serão passados para a função.

Ao utilizá-lo, ele executa a função original, substituindo o objeto especificado para o contexto de execução e o `Array` especificado para os argumentos da função. O diagrama e código a seguir mostram como podemos usá-lo para fazer o console imprimir todos os elementos de `argumentos` :

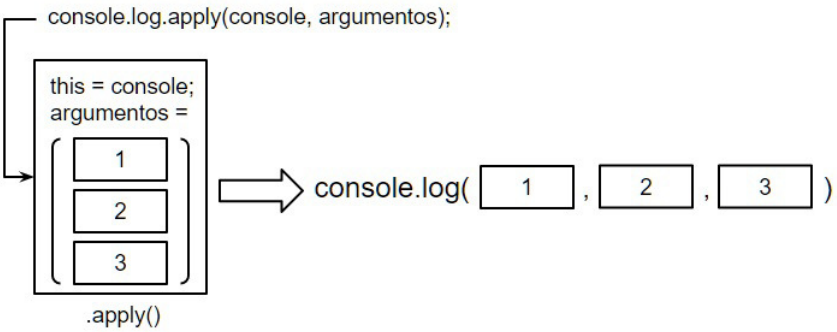


Figura 14.1: Método apply em ação

```
var argumentos = [1,2,3];
console.log.apply(console, argumentos);
// 1, 2, 3
```

Com o ES6, temos uma alternativa semelhante, porém otimizada. Nela não há a necessidade de lidar com o contexto de execução e temos a vantagem de poder tratar listas que são formadas dinamicamente, ou seja, onde não sabemos exatamente em qual índice está a informação que queremos.

Como você já deve suspeitar, estamos falando do operador Spread (também chamado de operador de propagação). Ele é bem semelhante ao operador Rest no que diz respeito à sua sintaxe, pois ambos utilizam a mesma notação de `...` (três pontos). No entanto, não se engane: eles funcionam de forma completamente opostas.

O operador Spread permite que uma expressão seja expandida onde múltiplos argumentos são esperados, enquanto o Rest faz o contrário. Ele comprime múltiplos argumentos em um `Array`. Vamos ver o que isso significa na prática.

Voltando para o nosso caso do `console.log`, podemos remover a função `apply` e simplesmente inserir como parâmetro do `log` os `...` (três pontos) acompanhado com o nome do `Array` que queremos que seja extraído — neste caso, a variável `argumentos`. O operador vai interpretar cada item contido em `argumentos` e passará para a função como se fossem parâmetros separados. Ao final, o resultado será equivalente a ter invocado `console.log(1,2,3)`, exatamente como na figura:

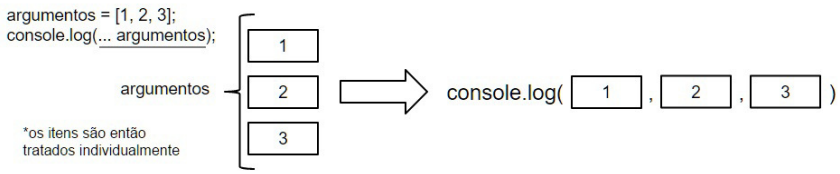


Figura 14.2: Operador Spread em ação

Aplicando o código, temos:

```
const argumentos = [1,2,3];
console.log(...argumentos) // 1, 2, 3
```

Pronto! Conseguimos fazer com que o `log` mostre todos os valores de `argumentos` em uma tacada só. Bem prático, não é mesmo? Mas o poder e utilidade deste operador não terminam aí. Veremos mais alguns exemplos para entender melhor como ele funciona e ver no que ele pode ser útil no dia a dia.

14.1 FAZENDO COMPRAS COM O SPREAD

Vamos estudar esta funcionalidade com mais uma situação

hipotética. Imagine que acabamos de sair do trabalho e recebemos por mensagem duas listas de compras pelo celular: uma de nossa mãe e outra de nossa esposa. Como somos pessoas de bom coração (e não queremos receber bronca em casa de nenhuma das duas), nós vamos até o mercado fazer as compras. Mas como não queremos perder tempo, faremos as duas compras ao mesmo tempo em um supermercado só.

Para facilitar o nosso caso de estudo, vamos considerar neste momento que as listas não possuem itens iguais. Em termos de código, teríamos as listas traduzidas em algo assim:

```
const listaMae = ['leite', 'ovos', 'papel'];
const listaNamorada = ['arroz','feijão', 'suco'];
```

Como as compras serão feitas simultaneamente, na realidade temos uma lista de compras só, o resultado da união delas. Podemos unificá-las facilmente usando a função `concat` — nativa do objeto `Array` — para então criar uma lista só:

```
const listaCompras = listaMae.concat(listaNamorada);
console.log(listaCompras); // ['leite', 'ovos', ..., 'suco']
console.log(listaCompras.length) // 6
```

Chegando ao mercado, pegamos o carrinho e buscamos os itens um a um. Prestes a ir ao caixa, lembramos de que também precisamos comprar alguns itens para o escritório: adesivos, canetas e fita adesiva. No final, a lista de compras que temos agora é composta de três outras listas, cada uma com seus itens distintos:

- Lista de compras da mãe;
- Lista de compras da namorada;
- Lista de compras do escritório.

Agora que temos três listas, em termos de implementação, como podemos facilmente criar uma lista que contenha todos os itens de cada uma? Uma possibilidade é utilizar a função `concat`

novamente, já que ela aceita múltiplos parâmetros:

```
const listaCompras = listaMae.concat(listaNamorada, listaEscritori
o);
```

Esta abordagem funciona sem problemas. Porém, com o ES6, temos outra abordagem possível: o operador Spread. Ele nos permite pegar um `Array` e propagar seus itens individualmente, exatamente como fizemos com os argumentos no `console.log`.

Isso significa que cada um dos itens que estão contidos nele é tratado de forma isolada, mas sem alterar a estrutura original da origem. No nosso exemplo, utilizando o operador Spread, a nossa lista final fica:

```
const listaCompras = [ ...listaMae, ...listaNamorada, ...listaEscr
itorio];
```

Para cada uma das listas, o operador a desfragmenta e a substitui na declaração pelos itens contidos nela. Em termos práticos, isso significa que a declaração anterior é transformada em:

```
const listaCompras = [
    'leite', 'ovos', 'papel',
    'arroz','feijão', 'suco',
    'adesivos', 'canetas', 'fita adesiva'
];
```

Repare que criamos um novo `Array` com o nome `listaCompras`, populado com cada um dos valores das listas: `listaMae`, `listaNamorada` e `listaEscritorio`. Não podemos confundir o que fizemos com isto:

```
const listaCompras = [listaMae, listaNamorada, listaEscritorio];
```

Por que não? Qual a diferença entre atribuições?

A diferença é que, com o operador Spread, os elementos das listas são tratados individualmente e inseridos na lista resultante. No segundo caso, estamos inserindo cada lista, com todos os seus

elementos, como sendo apenas um item da lista resultante. Em outras palavras, formamos uma lista de listas. Fica bastante claro quando imprimimos os seus conteúdos no console:

```
// com spread
console.log(listaCompras);
[
    'leite','ovos','papel',
    'arroz','feijão','suco'
    'adesivos', 'canetas', 'fita adesiva'
]

// sem o spread
console.log(listaCompras);
[
    ['leite','ovos','papel'],
    ['arroz','feijão','suco'],
    ['adesivos', 'canetas', 'fita adesiva'],
]
```

Agora que compreendemos que, com este operador, podemos tratar de forma individual múltiplos argumentos, vamos ver outras situações em que ele pode ser útil e como ele se difere do operador Rest.

14.2 ADICIONANDO ITENS A UM ARRAY

Como o operador Spread permite que trabalhemos individualmente com os múltiplos itens de uma lista, podemos utilizá-lo não somente para criar listas unificadas, mas também para unificar itens isolados com listas. Como por exemplo, em um *e-commerce* de uma franquia de roupas femininas, no qual podemos fazer uso nas páginas dos produtos para adicioná-lo no carrinho de compras (funcionalidade do botão "adicionar ao carrinho"):

```
const produtoSelecionado = {
    descricao: 'Blusa de Lã',
    preco: 'R$ 59,90'
};

const carrinho = [
```

```
    { descricao: 'Bota de Cano Médio', preco: 'R$ 199,90' },
    { descricao: 'Saia Colorida', preco: 'R$ 29,90' },
    { descricao: 'Vestido Longo', preco: 'R$ 399,90' },
];

const carrinhoAtualizado = [...carrinho, produtoSelecionado];
```

Para atualizar o carrinho, criamos um novo `Array` chamado `carrinhoAtualizado`, e nele propagamos `carrinho` e `produtoSelecionado`. O operador então processou todos os itens e inseriu-os no `carrinhoAtualizado`. Se dermos uma olhada no carrinho, veremos que todos os nossos produtos estão lá:

```
for(let produto of carrinhoAtualizado) {
    console.log(produto.descricao);
}

// saída:
// Bota de Cano Médio
// Saia Colorida
// Vestido Longo
// Blusa de Lã
```

14.3 OPERADOR SPREAD EM CHAMADAS DE FUNÇÕES

O uso mais comum e interessante desta nova funcionalidade é em conjunto a chamadas de funções, como por exemplo, se temos uma função `soma` que realiza soma de dois números. Podemos invocá-la assim:

```
function soma(a,b) {
  console.log(a + b);
}

soma(1,2); // 3
```

Como também podemos utilizar o operador:

```
function soma(a,b) {
  console.log(a + b);
}
```

```
var numeros = [1,2];
soma(...numeros); // 3
```

O resultado final é o mesmo. Mas não se limita só a isso. Repare como não é somente útil para objetos que sejam `Array`, mas para qualquer objeto que seja iterável.

No capítulo de iteradores e iteráveis, vimos que o JavaScript já possui alguns tipos que possuem esta característica, como: `Map`, `String` e `Set`. Para todos eles, podemos aplicar o operador Spread. Aplicando-o em uma `String`, por exemplo, podemos criar uma função chamada `contaQuantidadeVogaisNaoAcentuadas` que calcula a quantidade de vogais não acentuadas em uma palavra:

```
function contaQuantidadeVogaisNaoAcentuadas(palavra) {
  let qtdVogais = 0;
  const letras = [... palavra];
  for(let letra of letras) {
      if("aeiou".indexOf(letra) !== -1) {
          qtdVogais++;
      }
  }

  return qtdVogais;
}
```

Para não ter problemas com diferenças entre letras maiúsculas e minúsculas afetando a comparação, sempre deixamos as palavras em minúsculo antes de verificar suas vogais. Para isso, usamos a função `toLowerCase` da `String`.

```
function contaQuantidadeVogaisNaoAcentuadas(palavra) {
  let qtdVogais = 0;
  const palavraLowerCase = palavra.toLowerCase();
  const letras = [... palavraLowerCase];
  for(let letra of letras) {
      if("aeiou".indexOf(letra) !== -1) {
          qtdVogais++;
      }
  }

  return qtdVogais;
```

```
}
```

Agora, validando alguns casos, vemos que o código funciona perfeitamente para ambos os casos:

```
contaQuantidadeVogaisNaoAcentuadas('ecmascript'); // 3
contaQuantidadeVogaisNaoAcentuadas('javascript'); // 3
contaQuantidadeVogaisNaoAcentuadas('SWIFT'); // 1
contaQuantidadeVogaisNaoAcentuadas('jAvA'); // 2
```

Mas e se quisermos fazer com o que o nosso método conte a quantidade de vogais não acentuadas não somente de uma palavra, mas de uma quantidade indeterminada de palavras, textos e/ou frases? Será que é possível? Acredito que já vimos um outro operador que pode nos ajudar com isso... Porém, será que é possível utilizá-lo com o Spread?

14.4 OPERADOR REST X OPERADOR SPREAD

Apesar de ambos utilizarem a notação `...` (três pontos), o funcionamento deles é totalmente diferente:

- O operador Rest coleta os itens e coloca-os em um `Array`;
- O operador Spread torna um `Array` (e outros objetos iteráveis) em itens individuais.

Entretanto, isso não nos impede de usá-los em conjunto. Para ver como, primeiro alteramos o argumento do nosso método para aceitar não só um parâmetro, mas um número indefinido deles. Fazemos essa alteração com o operador Rest:

```
function contaQuantidadeVogaisNaoAcentuadas(...palavras) {
    // lógica da função
}
```

Agora alteramos a nossa lógica para contar as vogais não acentuadas, não somente de uma palavra, mas da quantidade de

palavras que serão enviadas no momento em que a função for invocada:

```
function contaQuantidadeVogaisNaoAcentuadas(...palavras) {
  let qtdVogais = 0;
  for(let palavra of palavras) {
    let palavraLowerCase = palavra.toLowerCase();
    const letras = [... palavraLowerCase];
    for(let letra of letras) {
      if("aeiou".indexOf(letra) !== -1) {
        qtdVogais++;
      }
    }
  }

  return qtdVogais;
}
```

E está feito! Agora podemos utilizar o `contaQuantidadeVogaisNaoAcentuadas` para validar uma, dez, cem ou mil palavras! A imaginação (e quantidade de memória disponível) são o limite!

```
contaQuantidadeVogaisNaoAcentuadas('es6'); // 1
contaQuantidadeVogaisNaoAcentuadas('java', 'javascript', 'delphi')
; // 7
contaQuantidadeVogaisNaoAcentuadas('NãO ConSideRa AcentUADoS'); //
 10
```

CAPÍTULO 15

DESESTRUTURAMENTO DE ARRAYS E OBJETOS

Como podemos notar com decorrer do livro, a principal proposta do ES6 é diminuir a quantidade de código JavaScript repetido que escrevemos e torná-lo mais simples de entender e dar manutenção. Já abordamos muitas funcionalidades que cumpriram esta meta, como:

- O laço de repetição `for...of` torna a iteração de objetos iteráveis simples;
- O `const` e o `let` tornam as variáveis melhores contextualizadas;
- Template Strings tornam a manipulação de Strings muito menos dolorosa;
- Arrow functions tornam a declaração de funções anônimas menos verbosa.

A funcionalidade de desestruturamento (*destructing*), tema deste capítulo, tem esta mesma finalidade. Resumidamente, podemos definir o desestruturamento como uma maneira de extrair valores armazenados em objetos e `Arrays`. A sua ideia é permitir que usemos a mesma sintaxe de construção de dados para extrair dados. Para entender o que isso significa, vamos utilizar o caso de uso do registro de um novo usuário em uma rede social.

Imagine que na nossa rede social fictícia, o devfriends.com.br, o

usuário precisa oferecer alguns dados para conseguir se cadastrar: nome, sobrenome, senha, e-mail, profissão e, claro, o mais importante, o link do perfil no GitHub. Estruturamos esses dados no objeto `Usuario` :

```
const Usuario = {
  nome: ' Elliot',
  sobrenome: 'Alderson',
  senha: 'mrrobot'
  email: 'elliot.alderson@gmail.com',
  profissao: 'Engenheiro de Cibersegurança'
  github: 'https://github.com/ElliotAlderson'
}
```

Antes que o usuário possa finalizar o seu cadastro, o sistema valida se o e-mail já não está sendo usado. Caso esteja, precisamos avisá-lo para que ele possa cadastrar um outro. Para fazer essa validação, temos de extrair a propriedade `e-mail` que está contida no objeto `Usuario` que foi construída a partir do registro feito na tela. Tradicionalmente, extrairíamos este valor e passaríamos para o método que faz a validação assim:

```
const email = Usuario.email;
validarEmail(email);
// ...
```

Com o desestruturamento de objetos do ES6, já podemos fazer diferente. O desestruturamento permite que usemos a sintaxe de objetos literais para extrair valores. Para extrair o `email` do `Usuario` , por exemplo, só precisamos fazer:

```
const {email} = Usuario;
console.log(email); // elliot.alderson@gmail.com
```

Note o que fizemos: do lado direito da atribuição, colocamos o nome dos campos que desejamos extrair entre `{ }` (chaves). Neste caso, colocamos somente `email` . Do outro lado, colocamos de onde queremos extrair estes valores. Neste cenário, o objeto `Usuario` .

Quando o código é executado, o valor de `Usuario.email` é extraído e atribuído para uma variável de mesmo nome do atributo, `email`. O efeito é o mesmo para extrair múltiplos valores de dentro do objeto:

```
const {senha, confirmacaoSenha} = Usuario;
console.log(senha); // mrrobot
console.log(confirmacaoSenha); // mrrobot
```

Entretanto, se tentamos extrair o valor de uma propriedade que não existe dentro do objeto `Usuario`, como uma propriedade chamada `numeroDoCartao`, por exemplo, não receberemos nenhum erro, mas a variável volta atribuída com `undefined`.

```
const {numeroDoCartao} = Usuario;
console.log(numeroDoCartao); // undefined
```

Note que a principal vantagem desta funcionalidade é que ela evita que percamos tempo com inúmeras declarações de variáveis, pois ela já faz esse trabalho automaticamente para nós quando extrai os dados da fonte. Mas podemos fazer muito mais. Vamos ver como.

15.1 ROTULANDO PROPRIEDADES

Nem sempre o nome da propriedade que extraímos de um objeto é bom ou claro o suficiente para nomear uma variável. Para reverter esta situação, podemos fazer uso de rótulos (*labels*). Assim, conseguimos associar qualquer nome para a variável que será criada com o valor da propriedade que queremos extrair.

Atribuir um rótulo é muito simples, basta seguir a sintaxe `<propriedade>:<label>` no desestruturamento. Por exemplo, considere o objeto `Pessoa` a seguir:

```
const Pessoa = {
    sobrenome: 'Alberto'
}
```

Supondo que não queremos utilizar `sobrenome` como variável, atribuímos um rótulo com outro nome, como `apelido`:

```
const {sobrenome:apelido} = Pessoa;
```

Feito isso, `sobrenome` não possui valor nenhum, somente `apelido`:

```
console.log(sobrenome); // undefined
console.log(apelido); // Alberto
```

15.2 DESESTRUTURAMENTO DE VÁRIOS OBJETOS

Até então, vimos como podemos desestruturar um objeto para extrair suas propriedades em múltiplas variáveis de uma maneira bem prática. Entretanto, é preciso estar atento a um detalhe bastante importante: o desestruturamento é sempre aplicado no primeiro objeto encontrado.

Precisamos ter cuidado quando queremos aplicar o desestruturamento em mais de um objeto por vez. Por exemplo, considere a descrição de um `suco` em um objeto literal e o método `descreveSuco`:

```
const suco = {
  sabor: 'uva',
  quantidade: '500ml'
}

function descreveSuco({sabor, quantidade}) {
  return `Este suco é de sabor ${sabor} e possui ${quantidade}`;
}

descreveSuco(suco); // Este suco é de sabor uva e possui 500ml
```

Note que, no método `descreveSuco`, obtemos as propriedades `sabor` e `quantidade` direitinho, porque passamos corretamente o objeto que queríamos. Ele foi o primeiro objeto encontrado que

foi passado como argumento na invocação do método. Se quisermos desestruturar dois ou mais objetos de uma vez, precisamos utilizar uma `,` (vírgula) como separador.

Vejamos a seguir onde, além do `suco`, o método `descreveSuco` recebe um objeto que descreve o tipo de doce:

```
const suco = {
  sabor: 'uva',
  quantidade: '500ml'
}

const doce = {
  tipo: 'açucar'
}

function descreveSuco({sabor, quantidade}, {tipo}) {
  return `Este suco é de sabor ${sabor} e possui ${quantidade} ado
cicado com ${tipo}`;
}

descreveSuco(suco, doce);
// 'Este suco é de sabor uva e possui 500ml adocicado com açucar'
```

Como separamos os argumentos, o interpretador conseguiu definir sem dificuldade qual desestruturamento pertence a qual objeto. E assim como fizemos com dois argumentos, o mesmo vale para três, quatro, e assim por diante.

15.3 DESESTRUTURAMENTO EM RETORNO DE CHAMADAS DE MÉTODOS

Um caso bem comum para desestruturamento é no retorno de chamadas de métodos, principalmente quando estamos trabalhando com JSON. Por exemplo, imagine que estamos consumindo uma API REST de um serviço de clima para criar um aplicativo mobile de previsão do tempo. Como nossa aplicação ainda está em versão beta, vamos implementar suas funcionalidades aos poucos.

O nosso primeiro objetivo é mostrar somente a temperatura atual em ºC (graus Celsius). Na documentação do serviço que vamos consumir, diz que o retorno da chamada que precisamos devolve um JSON que contém diversas informações:

- `temperatura` — Temperatura atual em ºC;
- `maxima` — Temperatura máxima em ºC;
- `minima` — Temperatura mínima em ºC;
- `descricao` — Descrição do tempo atual (por exemplo, nublado);
- E por aí em diante.

Em termos gerais, teremos isto:

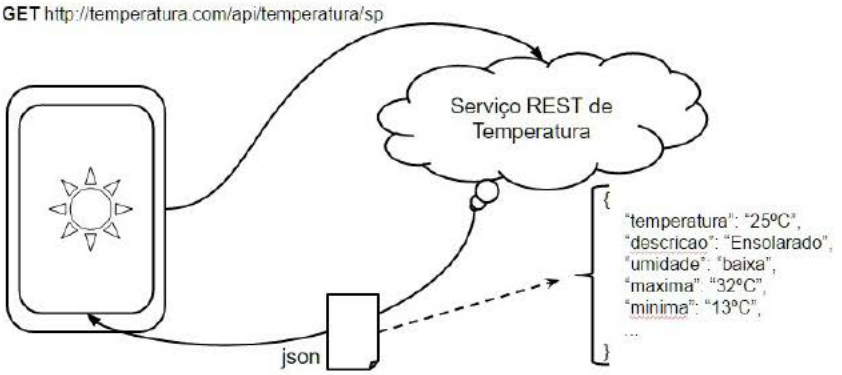


Figura 15.1: Funcionamento do aplicativo

Por ora, precisamos somente do dado da temperatura. Na nossa aplicação, podemos criar o método `recuperaDadosDaAPI` que faz uma requisição para o serviço externo e aguarda o resultado dela — o JSON com os dados solicitados:

```
function recuperaDadosDaAPI() {
  // Realiza GET para o http://temperatura.com/api/temperatura/sp
  // Retorna JSON que veio na response
  return response.json;
}
```

Como só usaremos a informação da temperatura, podemos utilizar o desestruturamento para obter somente o dado que queremos direto da resposta da chamada do método. Neste caso, o dado `temperatura`:

```
const {temperatura} = recuperaDadosDaAPI();
console.log(temperatura); // 24ºC
```

Com o desestruturamento, obtemos facilmente somente o que nos interessa em um único passo, deixando evidente para qualquer outro desenvolvedor quais os dados que estamos extraindo. Pensando nas próximas iterações do projeto, quando for necessário extrair mais informações do JSON, bastará colocar o nome da nova propriedade necessária entre `{}` (chaves).

```
const {temperatura, maxima, minima} = recuperaDadosDaAPI();
console.log(temperatura); // 24ºC
console.log(maxima); // 32ºC
console.log(minima); // 20ºC
```

Assim que fizemos com `temperatura`, conseguimos extrair `maxima` e `minima` sem problemas.

15.4 DESESTRUTURAMENTO DE ARRAYS

Outro ponto alto do desestruturamento fica evidente quando trabalhamos com `Array`. Imagine que temos um `Array` com uma simples lista de cores:

```
const cores = ['azul', 'cinza', 'branco', 'preto'];
```

Para pegar somente a primeira e a última cor da lista, por exemplo, temos de obter por índice:

```
const cores = ['azul', 'cinza', 'branco', 'preto'];

console.log(cores[0]); // azul
console.log(cores[3]); // preto
```

Mas esta maneira ficou no passado, agora podemos utilizar o

desestruturamento. Para isso, atribuímos uma variável na posição correspondente do `Array` do valor que queremos recuperar e deixamos os demais espaços vazios. Usaremos as variáveis `cor1` e `cor2` para extrair a primeira e última cores, respectivamente:

```
const [cor1,,,cor2] = ['azul', 'cinza', 'branco', 'preto'];

console.log(cor1); // azul
console.log(cor2); // preto
```

Se tentamos recuperar um valor que não existe dentro dos limites do `Array`, como um elemento no índice 5, logo depois da `cor2`, não recebemos um erro, mas a variável é atribuída com `undefined`:

```
const [cor1,,,cor2, cor3] = ['azul', 'cinza', 'branco', 'preto'];

console.log(cor1); // azul
console.log(cor2); // preto
console.log(cor3); // undefined
```

Perceba que, como a variável `cor3` corresponde a um índice fora dos limites do `Array`, ela ficou como `undefined`. Bastante promissor, não acha?

Mas, além disto, esta funcionalidade fica ainda melhor quando combinamos os conceitos e criamos lógicas mais complexas. Para exemplificar, vamos partir para outro exemplo. Neste novo caso, leve em conta a seguinte lista de contatos da nossa agenda:

```
const contatos = [
  {
    nome: 'Alex Júnior',
    numero: '1234-5678'
  },
  {
    nome: 'Carolina Moya',
    numero: '1234-6789'
  },
  {
    nome: 'Fernando Jorge',
    numero: '1234-5567'
```

```
  }
];
```

Agora, como fazer para extrair somente o número de telefone do contato do meio, de nome "Carolina Moya"? Podemos aplicar a mesma lógica que aplicamos na lista de cores para somente pegar o segundo item da lista. E como queremos somente o número, vamos também criar uma função chamada `mostraNumero` que recebe um objeto que representa um contato da agenda e extrai somente a propriedade `numero`:

```
const [,Carol] = contatos;

function mostraNumero({numero}){
  console.log(numero);
}

mostraNumero(Carol); // 1234-6789
```

O código funciona, mas podemos ir além:

```
function mostraNumeroDaCarol([,{numero}]){
  console.log(numero);
}

mostraNumeroDaCarol(contatos); // 1234-6789
```

Em vez de extrair a variável dos contatos e passar para o método `mostraNumero`, que então extrai o número do objeto, fizemos um método que já faz tudo isso. O `mostraNumeroDaCarol` já espera um `Array` onde extrairá o número do objeto da segunda posição na lista. Este número então é atribuído à variável e enviado para o console. No final, bastou chamar o método passando `contatos` como parâmetro.

Bem bacana, não?

15.5 DESESTRUTURANDO ARRAYS — PARTE 2

Neste exemplo anterior, extraímos o segundo objeto da lista

`contatos`, e então extraímos a propriedade `numero`. Mas e se por um acaso quisermos fazer o contrário? Será que é possível? Por exemplo: imagine que precisamos extrair qual é a primeira rota da lista `rapidas` contida no objeto `rotas`:

```
const rotas = {
  rapidas: ['Rodovia', 'Estrada X', 'Estrada Y']
};
```

Existem inúmeras maneiras de fazer isto, mas utilizando o que acabamos de aprender, fica assim:

```
const { rapidas:[rapida] } = rotas;
console.log(rapida); // Rodovia
```

Repare que sagaz o que fizemos: dentro das `{}` (chaves), indicamos a propriedade `rapidas` que queríamos extrair. Como essa propriedade é um `Array`, conseguimos extrair imediatamente o primeiro objeto indicando a variável `rapida` entre `[]` (colchetes). Na execução do método, tivemos o resultado esperado: `Rodovia`.

Deu um nó na cabeça? Não se preocupe, a princípio realmente dá. Mas o conceito é realmente simples: sempre mostramos quem é o objeto alvo e qual propriedade queremos obter dele. Releia os exemplos e certifique-se de que compreendeu cada passo. Faça os exercícios e, como mágica, você verá que esta sintaxe é na verdade bem simples.

CAPÍTULO 16

MODELANDO COM CLASSES

Uma das maiores dificuldades ao estudar JavaScript é entender a maneira como herança funciona, pois ela não funciona da maneira como aprendemos em Orientação a Objetos (OO), ou em linguagens como o Java e C#. O JavaScript possui herança por prototipagem (*Prototypal Inheritance*).

A ideia por trás da funcionalidade de classes no ES6 é que possamos criar uma hierarquia de objetos que, por debaixo dos panos, ainda funciona em cima da herança por prototipagem. Classes formam um novo paradigma que atende como uma camada de abstração em cima deste tipo de prototipagem, tornando o código e a aprendizagem mais fáceis e próximos ao que já conhecemos como Orientação a Objetos.

Para este capítulo, falaremos sobre carros. Imagine que somos os selecionados para criar um sistema de gerenciamento das máquinas de uma fábrica de carros. Iniciaremos o desenvolvimento modelando o produto principal da fábrica, o carro. Todos os carros montados na fábrica possuem características em comum:

- Modelo;
- Chassi;
- Quantidade de portas;
- Tipo de combustível;

- Tração;
- Transmissão.

Para tornar nosso exemplo mais didático, vamos usar somente as três primeiras propriedades: modelo, chassi e quantidade de portas. Vamos criar uma função construtora para criar um objeto `Carro` que contenha estas propriedades:

```
function Carro(modelo, chassi, qtdPortas) {
  this.modelo = modelo;
  this.chassi = chassi;
  this.qtdPortas = qtdPortas;
}
```

Agora que temos um modelo de `Carro` como base, já podemos instanciar um objeto protótipo dele:

```
const prototipo = new Carro('protótipo', '1290381209', 2);
console.log(prototipo.modelo); // protótipo
console.log(prototipo.chassi); // 1290381209
console.log(prototipo.qtdPortas); // 2
```

Agora que temos os atributos básicos, vamos adicionar algumas funcionalidades. Todo carro deve ter a capacidade de andar, afinal, esta é a sua finalidade: transportar pessoas e objetos. Vamos adicionar esta capacidade com o método `andar` pela propriedade `prototype` do nosso `Carro`. Isso faz com que todas as instâncias de `Carro` tenham esta capacidade de andar também:

```
function Carro(modelo) {
  this.modelo = modelo;
}

Carro.prototype.andar = function() {
  console.log("vrum vrum");
}

const prototipo = new Carro('protótipo', '1290381209', 2);
prototipo.andar(); // vrum vrum
```

Agora que temos o modelo básico do carro da fábrica junto a sua função primária, podemos instanciar o primeiro modelo oficial

da montadora, o `Sonix` .

```
const sonix = new Carro('Sonix', '9120219', 4);
console.log(sonix.modelo); // Sonix
sonix.andar(); // vrum vrum
```

Nosso carro como produto está modelado e já anda, mas nosso modelo ainda está incompleto. O modelo Sonix é diferenciado e possui várias outras funcionalidades e atributos que os modelos de carro não possuem. Para que possamos fazer isto no nosso modelo de fábrica, o próximo passo é tornar o "Sonix" uma entidade separada de `Carro` .

Vamos estender este objeto `Carro` , de modo que tornaremos `Sonix` um objeto que herde `Carro` . Ou seja, compartilhará de todas as funcionalidades e atributos que o carro tem, mas terá suas próprias características. Vamos começar fazendo a extensão do `Sonix` utilizando a herança por prototipagem:

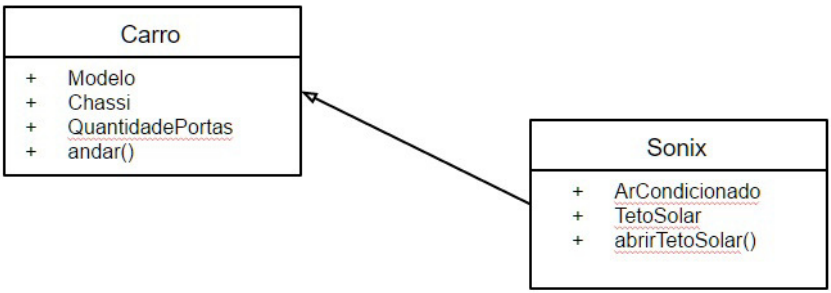


Figura 16.1: Relação entre os modelos de carros

```
function Sonix(modelo, chassi, qtdPortas) {
  Carro.call(this, modelo, chassi, qtdPortas);
}

Sonix.prototype = Object.create(Carro.prototype);
Sonix.prototype.constuctor = Sonix;

const sonix = new Sonix('Sonix', '9120219', 4);
console.log(sonata.modelo); // Sonix
sonata.andar(); // vrum vrum
```

Agora nosso modelo `Sonix` possui todos os atributos e métodos de `Carro`. O próximo passo é ir além e atribuir os métodos e atributos exclusivos deste modelo de carro. No nosso caso, este modelo é diferenciado no mercado por possuir teto solar. Vamos implementar esta característica novamente usando `prototype`:

```
Sonix.prototype.abrirTetoSolar = function() {
    console.log('abrindo teto solar');
}

const sonix = new Sonix('Sonix', '9120219', 4);
sonix.abrirTetoSolar(); // abrindo teto solar
```

Muito bem, conseguimos criar um modelo básico que representa todos os carros construídos da fábrica, e então estendemos este modelo para um específico que herda estas características básicas, mas possui os seus atributos próprios. Assim como fizemos com o modelo `Sonix` da nossa fábrica fictícia, o mesmo procedimento seria feito para os inúmeros modelos de carro da fábrica.

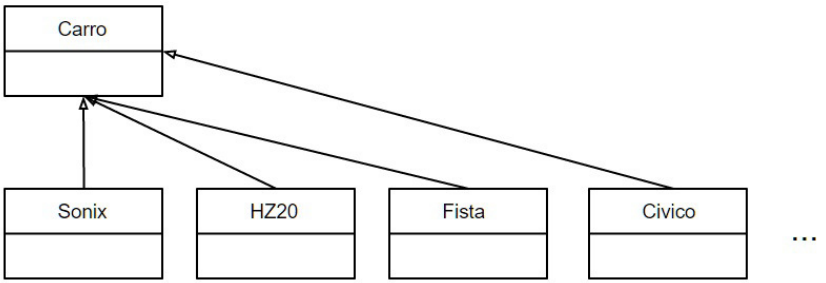


Figura 16.2: Modelo da Fábrica de Carros

Conseguimos completar nossa missão de modelar os carros, mas repare como isso não foi nem um pouco trivial. Para conseguir estender os modelos de carros, tivemos de usar o método `call` na função construtora e o `Object.create` no atributo `prototype`. Essa abordagem é extremamente confusa para quem está

escrevendo o código tanto para quem vai dar manutenção nele depois.

É aí que o ES6 entra em ação novamente.

16.1 UTILIZE CLASSES DO ES6

Para mostrar como a nova sintaxe de classes torna o código muito mais limpo e descritivo, vamos refatorar o exemplo da fábrica de carros usado no tópico anterior para a nova sintaxe. Lembrando, mais uma vez, de que, apesar do modo como vamos implementar a hierarquia no código ser bem diferente, por debaixo dos panos continua sendo herança por prototipagem. É somente uma nova abstração para facilitar o nosso entendimento.

Para começar, criaremos nosso `Carro` em uma classe JavaScript usando a nova sintaxe. Antes, criamos nosso `Carro` utilizando uma função construtora. Agora faremos diferente, usaremos a palavra reservada `class`:

```
class Carro {
  // implementação do carro
}
```

Agora que temos o carro, precisamos inserir um construtor. O construtor nos permitirá inserir os atributos dentro do `Carro`. No caso: `modelo`, `chassi` e `qtdPortas`. Com a finalidade de fazer isto, usamos a palavra reservada `constructor`:

```
class Carro {
  constructor(modelo, chassi, qtdPortas){
      this.modelo = modelo;
      this.chassi = chassi;
      this.qtdPortas = qtdPortas;
  }
}
```

Muito bem. Além destes atributos, nosso modelo de carro básico

também precisa da função `andar`. Então, vamos inserir a função na classe usando a mesma sintaxe que vimos no capítulo *Melhorias em objetos literais*. Ou seja, sem utilizar a palavra reservada `function`, nem os dois pontos, nem o objeto `prototype`. Vamos simplesmente escrever a nossa função dentro da classe:

```
class Carro {
  constructor(modelo, chassi, qtdPortas){
    this.modelo = modelo;
    this.chassi = chassi;
    this.qtdPortas = qtdPortas;
  }

  andar() {
    console.log("vrum vrum");
  }
}
```

Note que, ao contrário do que fizemos no ES5, na sintaxe de classes não há necessidade de separar os métodos com `,` (vírgula). Aliás, se fizermos isso por força do costume, tomaremos um erro de sintaxe `Unexpected ,`. Agora que temos nossa classe definida, podemos instanciá-la:

```
const basico = new Carro('Básico', '123123', 2);

console.log(basico.modelo); // Básico
console.log(basico.chassi); // 123123
console.log(basico.qtdPortas); // 2
basico.andar(); // vrum vrum
```

Bem bacana esta nova sintaxe, não? Olhe como ela é muito semelhante ao que já estamos acostumados a usar em linguagens orientadas a objeto. Além disso, o código parece mais transparente, limpo e fácil de entender.

Mas calma, ainda não acabamos nosso trabalho. No exemplo anterior, criamos uma classe que estendia `Carro`, a classe `Sonix`. Faremos isso utilizando a nova sintaxe.

16.2 ESTENDENDO CLASSES

Como você já deve ter imaginado, para estendermos uma classe, usamos a palavra reservada `extends`. Para iniciar, declaramos a classe `Sonix` com as suas características próprias, como fizemos com `Carro`:

```
class Sonix {
  abrirTetoSolar() {
    console.log('abrindo teto solar');
  }
}
```

Agora, tornamos `Sonix` uma extensão de `Carro` com a palavra reservada `extends`:

```
class Sonix extends Carro {
  abrirTetoSolar() {
    console.log('abrindo teto solar');
  }
}
```

Pronto! Já conseguimos estender `Carro` na classe `Sonix`. Não acredita em como foi simples? Então tire a prova você mesmo. Ao instanciarmos um objeto da classe `Sonix`, podemos ver que ele consegue executar os métodos `andar` — provida de `Carro` e `abrirTetoSolar`.

```
const sonix = new Sonix();
sonata.abrirTetoSolar(); // abrindo teto solar
sonata.andar(); // vrum vrum
```

Mas como você já deve suspeitar, apesar de parecer que está tudo certo, é necessário estar atento aos detalhes. Quando estendemos uma classe, emprestamos todos os métodos e propriedades da classe pai para as classes filhas.

No exemplo, apesar do método `andar` estar funcionando, as propriedades da classe `Carro` (`modelo`, `chassi` e `qtdPortas`) não estão funcionando na classe `Sonix`. Precisamos corrigir isso.

Precisamos criar um construtor em `Sonix` que aceite estes atributos. Feito isso, precisaremos chamar o construtor da classe pai, com a palavra reservada `super`, para que ele configure corretamente a propriedade na classe. Vamos lá:

```
class Sonix extends Carro {
  constructor(modelo, chassi, qtdPortas) {
    super(modelo, chassi, qtdPortas);
  }

  abrirTetoSolar() {
    console.log('abrindo teto solar');
  }
}
```

Muito bem. Agora nosso construtor recebe os parâmetros e passa-os para o construtor da classe pai. Agora, ao testar nossa classe, notaremos que está tudo como planejado.

```
const sonix = new Sonix('Sonix', '9120219', 4);
sonata.abrirTetoSolar(); // abrindo teto solar
sonata.andar(); // vrum vrum
console.log(sonata.modelo); // Sonix
```

16.3 HOISTING EM CLASSES

Agora que aprendemos como a sintaxe funciona, precisamos nos atentar a alguns detalhes importantes. Um deles é que, no ES5, quando criamos objetos por meio de funções, temos o comportamento de *hoisting*. Isto é, dentro de um escopo, as funções que são declaradas são imediatamente disponíveis para uso — independente de onde as declarações acontecem.

Em outras palavras, isso significa que podemos chamar uma função que é declarada só depois no código. Veja o exemplo a seguir, no qual conseguimos instanciar `Carro` antes mesmo de declará-lo:

```
const carro = new Carro('sonix');
console.log(carro.modelo);
```

```
function Carro(modelo) {
  this.modelo = modelo;
}
```

Porém, declarações de classes não possuem este comportamento. A classe só existe quando a execução do código chega ao ponto onde a classe é avaliada. Se tentarmos acessar antes, tomaremos um `ReferenceError` .

```
const carro = new Carro('sonix'); // ReferenceError
class Carro {
  constructor(modelo) {
    this.modelo = modelo;
  }
}
```

16.4 DECLARAÇÃO POR MEIO DE EXPRESSÕES

Apesar de ser incomum, também podemos declarar nossas classes por expressões, assim como fazemos com funções. Do mesmo modo, elas podem ser anônimas, como no exemplo a seguir:

```
const Classe = class {
    // ...
};

const instancia = new Classe();
```

Ou então podemos nomeá-la localmente:

```
const Classe = class Nome {
    getClassName() {
        return Nome.name;
    }
};

const instancia = new Nome();
```

Se tentarmos acessar a classe por seu nome interno, `Nome` , não conseguiremos.

```
console.log(instancia.getClassName()); // Nome
console.log(Nome.name); // ReferenceError: Nome is not defined
```

16.5 MÉTODOS ESTÁTICOS

A nova sintaxe de classes do ES6 nos permite declarar métodos estáticos na definição de nossas classes. Este tipo de método pode ser invocado sem que seja instanciado um novo objeto da classe. Para definir este tipo de método, usamos a palavra reservada `static`.

No próximo exemplo, temos uma função `abrirPorta` na classe `Casa` que pode ser invocada sem a necessidade de instanciar um novo objeto (`new`). Basta usar o nome da própria classe para chamar a função.

```
class Casa {
  static abrirPorta(){
    console.log('abrindo porta');
  }
}
Casa.abrirPorta(); // abrindo porta
```

Note que, ao remover o `static`, ao tentar executar o código, receberemos um erro.

```
TypeError: Casa.abrirPorta is not a function
```

16.6 ATRIBUTOS PRIVADOS COM WEAKMAP

Atualmente, a sintaxe de classe do JavaScript ES6 não dá suporte a propriedades privadas. Mas como vimos no capítulo de Mapas, podemos utilizar a estrutura de `WeakMap` para nos ajudar.

Para exemplificar, vamos considerar uma classe `Videogame` com alguns atributos: nome, número de controles, tipo de saída de vídeo e mídia. Esconderemos estes atributos usando um `WeakMap`.

```
const propriedades = new WeakMap();

class VideoGame {
    constructor(nome, controles, saida, midia) {
        propriedades.set(this, {
          nome, controles, saida, midia
        });
    }
}
```

Ao instanciar um `VideoGame` e tentar acessar qualquer um de seus atributos, não conseguimos.

```
const caixa360 = new VideoGame('caixa360', 4, 'hdmi', 'dvd');
console.log(caixa360.nome); // undefined
```

Para este caso, é preciso expor métodos que tornem possível acessar estes atributos. Então, criaremos uma função `recuperaPropriedade` que recebe como parâmetro o nome da propriedade que queremos e devolve o seu valor, caso exista. Caso contrário, recebemos `undefined`.

```
const propriedades = new WeakMap();

class VideoGame {
    constructor(nome, controles, saida, midia) {
        propriedades.set(this, {
          nome, controles, saida, midia
        });
    }

    recuperaPropriedade(propriedade) {
      return propriedades.get(this)[propriedade];
    }
}
```

Agora sim é possível recuperar as propriedades da classe `VideoGame`.

```
const caixa360 = new VideoGame('caixa360', 4, 'hdmi', 'dvd');
console.log(caixa360.recuperaPropriedade('nome')); // caixa360
```

16.7 ÚLTIMAS CONSIDERAÇÕES

Neste capítulo, espero tê-lo convencido das vantagens da nova sintaxe de classes do JavaScript. Muitos vão ficar decepcionados pela impossibilidade de ter propriedades privadas nativamente nas classes, mas é necessário lembrar de que o JavaScript continua trabalhando com herança por prototipagem por debaixo dos panos.

Para não dizer que não há esperança, existe uma proposta formal no repositório da TC39 da funcionalidade de campos privados nativos nas classes. Pode ser que ela seja implementada nas versões futuras do ECMAScript. A proposta pode ser encontrada no repositório do GitHub, em https://github.com/tc39/proposal-private-fields.

No próximo capítulo, vamos aprender como tirar ainda mais proveito desta sintaxe para modularizar os nossos projetos e deixá-los ainda melhores.

Capítulo 17

MÓDULOS

Quando estamos trabalhando com projetos relativamente pequenos — como projetos pessoais ou para a faculdade —, lidar com a complexidade de uma aplicação não costuma ser uma tarefa muito árdua. Não costuma existir muitos arquivos nestes projetos, então acabamos dando o nosso jeito e conseguimos nos organizar da nossa maneira.

Entretanto, quando começamos a trabalhar em projetos reais, de empresas grandes, vemos que a realidade é totalmente diferente. A complexidade nestes projetos é muito maior: existem milhares de arquivos e linhas de códigos que precisam ser administrados. É aí que a brincadeira fica séria.

Uma das estratégias consagradas em desenvolvimento de software para lidar com complexidade de código é a da modularização. Em termos simples, modularizar significa dividir o código em partes que representam uma abstração funcional e reaproveitável da aplicação. Modularizar um código permite que consigamos não somente organizá-lo, mas também reutilizá-lo. E isso se aplica a qualquer projeto e/ou linguagem de programação.

Pense em um sistema de controle e gerenciamento de horas de trabalho, como o consagrado Jira (https://www.atlassian.com/software/jira) da Atlassian. O requisito funcional mais importante deste sistema é permitir que os funcionários de uma empresa, cada um com seu login e senha

pessoais, loguem e consigam registrar suas horas de trabalho. Pensando neste tipo de sistema, logo de cara já podemos imaginar duas telas para ele:

- Tela de login;
- Tela para registro de horas.

Ao pensar nas funcionalidades essenciais destas telas, já podemos fazer uso da estratégia de dividir a implementação em módulos. Em vez de implementar todas as lógicas, funções e variáveis em um único arquivo JavaScript, podemos separá-las em responsabilidades diferentes. Ter um código que seria responsável pela parte de login, outro pela parte de cadastro e assim em diante. O diagrama a seguir mostra uma possível estratégia que poderíamos adotar para modularizar este sistema:

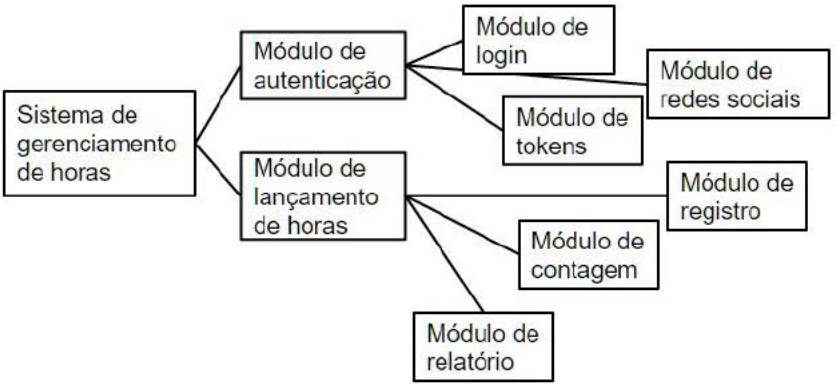


Figura 17.1: Simulação de modularização de um sistema

Um dos grandes problemas do JavaScript, até a versão 5.1 do ECMAScript, é que ele não dava suporte nativo a módulos e isso dificultava muito a manutenção de projetos complexos. Como a necessidade por modularizar foi se tornando cada vez maior, com o tempo surgiram algumas alternativas para conseguir se fazer isso.

Tornou-se necessário utilizar bibliotecas externas que implementam especificações, como o *CommonJS* (http://requirejs.org/docs/commonjs.html) e *AMD* (Asynchronous Module Definition). Entretanto, baseado fortemente no ecossistema Node e na linguagem Python, e com a meta de agradar aos usuários de ambas especificações, o ES6 trouxe esta funcionalidade valiosa para o JavaScript.

Módulos nada mais são que funcionalidades do código dividido em partes. Cada módulo compõe peças do código que podem ser adicionadas e removidas em um projeto sempre que necessário. Com esta abordagem, temos a redução da complexidade, separação de responsabilidades — já que força o desenvolvedor a determinar limites e responsabilidades para cada módulo —, manutenção facilitada e, principalmente, reutilização do código.

Vamos modularizar na prática.

17.1 COMMONJS X AMD

A fim de conseguir trabalhar com módulos no JavaScript, ao longo dos anos foram criados vários padrões. Dentre tantos criados, os que ganharam maior aceitação na comunidade foram dois: o CommonJS e AMD. Vejamos rapidamente um pouco sobre cada um.

CommonJS

O CommonJS estrutura uma API síncrona para trabalhar com módulos no JavaScript. É uma abordagem que funciona muito bem, principalmente do lado do servidor — estrutura para qual a especificação foi projetada, como é o caso da sua implementação mais famosa: o Node.js.

Sem entrar em muitos detalhes, já que isso daria assunto para mais um outro livro inteiro, resumiremos como ela funciona. No caso do Node.js, quando precisamos utilizar um módulo (próprio ou não), usamos a função reservada `require`. Esta aceita um parâmetro que é o nome do módulo que queremos carregar.

Um exemplo clássico para importação de módulos é a utilização da biblioteca Express.js (http://expressjs.com/) para subir um servidor. Para importar o módulo `express` para subir nossa aplicação, precisamos de um código semelhante ao exemplo a seguir.

```
var express = require('express');
var app = express();
app.listen(8080, function() {
    console.log('Hello World');
});
```

Ao usarmos o `require`, o Node.js se encarrega de importar o módulo para dentro do arquivo. Feito isso, já é possível utilizar os métodos disponíveis no módulo, como é feito na variável `app`, na qual é usada a função `express` e `listen` para subir o servidor.

Quando esta abordagem é levada para o lado do cliente, o funcionamento é um pouco diferente. É necessário usar ferramentas de build pré-produção. Nestes casos, utilizamos ferramentas de front-end como o Browserify (http://browserify.org/) e o Webpack (https://webpack.github.io/). Elas fazem o trabalho de buscar as dependências entre os módulos e juntá-los em um único arquivo que fica disponível para o front-end.

AMD

O AMD (acrônimo para *Asynchronous Module Definition*) também define uma API com um mecanismo bastante robusto para definição de módulos e dependências, assim como o CommonJS, com a diferença de que, no AMD, o mecanismo é assíncrono. Este

padrão foi desenvolvido especialmente para ambientes do lado do cliente, ou seja, para ambientes dos navegadores.

Muitas bibliotecas implementam este padrão, sendo a mais reconhecida e utilizada a RequireJS (http://requirejs.org/). O RequireJS pode ser usado para importar o JQuery para dentro de uma página, por exemplo. Para isso, é preciso definir as configurações do que precisaremos em um arquivo, utilizando o método `requirejs.config` :

```
requirejs.config({
    "baseUrl": "js/modulos",
    "paths": {
        "app": "../app",
        "jquery": "//code.jquery.com/jquery-3.1.1.min.js",
    }
});
```

Neste exemplo, definimos que o `jquery` seja baixado da URL especificada e disponibilizada no diretório `app` , através do seu caminho relativo. Então, para utilizá-la na página, empregamos o método `define` . Neste método, definimos o nome dos módulos que queremos importar e uma função de `callback` que indica como este módulo será usado.

```
define(["jquery"], function($) {
    $(function() {
        $('body')... // já podemos usar os seletores do jQuery, po
r exemplo
    });
});
```

Este exemplo pode ser visto na íntegra na página oficial da biblioteca, em http://requirejs.org/docs/jquery.html.

17.2 IMPORTAR E EXPORTAR MÓDULOS

Com o ES6, tudo o que temos no JavaScript agora é interpretado como um módulo. Podemos modularizar desde uma variável, uma

função, até mesmo uma classe inteira. Cada módulo é armazenado em um arquivo JavaScript. Para que os módulos sejam compartilhados entre os diversos arquivos do código, precisamos utilizar as palavras reservadas `import` e `export` para importá-los e exportá-los, respectivamente.

Quando falamos em exportar um módulo, existem dois tipos que devem ser considerados: o padrão e o nomeado. O primeiro é para o caso em que temos algum valor primário (função, variável etc.). Este tipo só pode ser usado uma vez por módulo. Já o tipo nomeado é para quando precisamos que o nosso módulo possa ser consumido em pedaços, diretamente pelos nomes dos valores exportados. Este pode ser usado várias vezes por módulo.

Para analisar isto na prática, imagine que estamos criando uma aplicação web de geometria analítica, algo semelhante ao GeoGebra (https://www.geogebra.org/), projeto de código aberto de matemática dinâmica. Esta aplicação permite criar e aplicar diversas fórmulas e cálculos em cima de formas geométricas no plano cartesiano bidimensional e do plano tridimensional.

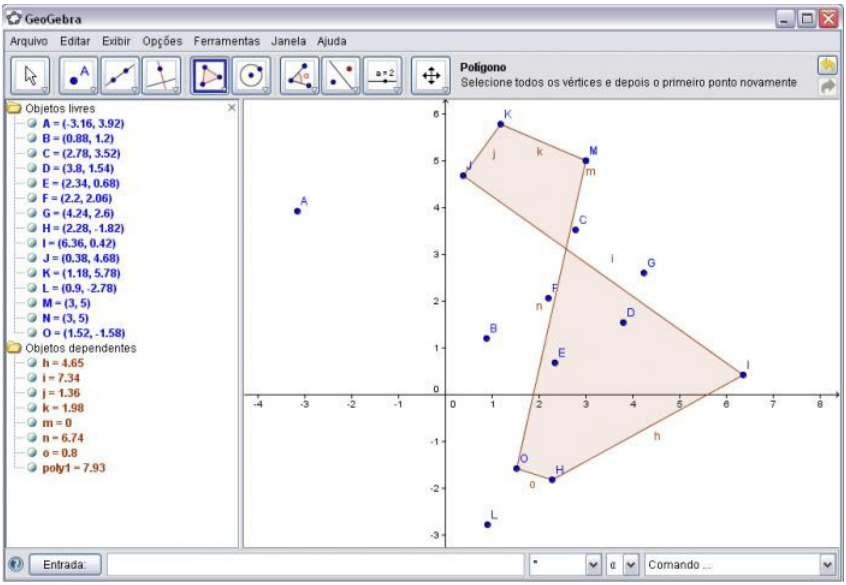


Figura 17.2: Interface do Software Geogebra

Considere que, neste projeto semelhante ao GeoGebra, começamos a trabalhar com circunferências. Para separar as operações de circunferência, criamos um módulo específico nomeado `circunferencia.js` . Neste módulo, definimos duas operações básicas de circunferência: `comprimento` e `area` . Como ambas envolvem o valor de PI, também criamos uma constante com o seu valor aproximado (~3.14):

```
const PI = 3.14;

function comprimento(raio) {
  return 2 * PI * raio;
}

function area(raio) {
  return PI * (raio * raio);
}
```

Agora que definimos um módulo, precisamos expô-lo para que os demais módulos da aplicação possam enxergar e utilizá-lo. Para

exportar as funções `comprimento` e `area` e a constante `PI`, usamos a função `export`.

```
export const PI = 3.14;

function comprimento(raio) {
  return 2 * PI * raio;
}

function area() {
  return PI * (raio * raio);
}

export default comprimento; // padrão
export {area, PI}; // nomeado
```

Repare que utilizamos os dois tipos de exportação neste módulo. Para a função `comprimento`, usamos a exportação padrão, utilizando a palavra reservada `default` logo após de `export`. Para os demais, usamos o tipo de exportação nomeado, apenas com o `export`.

A diferença entre os dois tipos de exportação está na hora de importar estas funções em outros módulos. Imagine que precisamos utilizar o módulo anterior em outro `matematica.js`. Se tivéssemos feito a importação do módulo sem definir os nomes das funções e/ou métodos que queremos importar explicitamente, teríamos somente acesso ao método `comprimento`.

```
import comprimento from './circunferencia';

comprimento(2); // ~ 12,5
```

Isso acontece porque definimos o método `comprimento` como padrão de importação do nosso módulo. Repare também que, quando utilizamos o `import`, não precisamos colocar a extensão do arquivo (`.js`). Se quisermos ter acesso a tudo que foi exportado deste módulo, temos de especificar.

```
import {comprimento, somar, PI} from './circunferencia';
```

```
comprimento(2); // 12,5
raio(2);  // ~ 12,5
console.log(PI); // 3.14
```

Também podemos usar o `*` (asterisco) para importar tudo que o módulo expõe. O único porém é que o asterisco sempre nos retorna um objeto. Por este motivo, é necessário definir uma variável na qual ele será alocado antes que possamos usá-lo. No trecho a seguir, usamos `circunferencia` como esta variável.

```
import * as circunferencia from './circunferencia';

circunferencia.comprimento(2); // ~ 12,5
circunferencia.raio(2);  // ~ 12,5
console.log(circunferencia.PI); // 3.14
```

Agora já podemos utilizar todos os recursos expostos do módulo `circunferencia.js`.

17.3 EXPORTANDO CLASSES

Assim como fizemos com funções e variáveis, também podemos exportar classes inteiras como módulos. Um caso bem interessante deste uso é na biblioteca de componentes do Facebook que vem ganhando muita força na comunidade JavaScript — e que já dá suporte ao ES6, o React (https://facebook.github.io/react/).

Explicando de forma bem sucinta, esta biblioteca permite fragmentar a nossa página HTML em componentes independentes que conversam entre si. Em outras palavras, em vez de definir no HTML a página inteira de uma vez, criamos pequenos componentes que compõe o todo. Veja a figura:

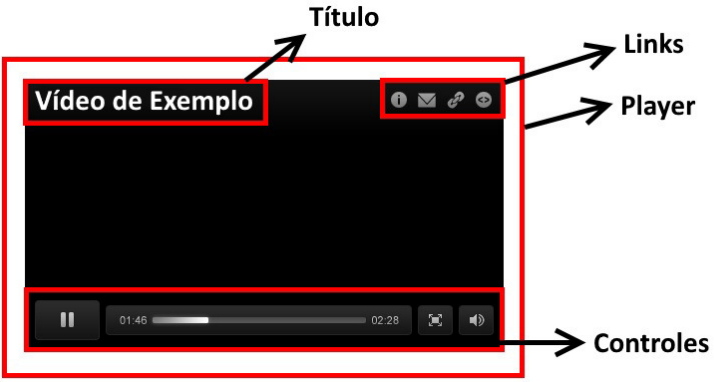


Figura 17.3: Exemplo de componentização de um player

Assim como dividimos o player em vários componentes na imagem, cada componente representa uma classe diferente, ou seja, um módulo. Para montar o player completo, precisamos importar todas outras classes para dentro dele. O título, links e controles teriam um código semelhante para exportar suas declarações.

```
class Controles {
    // implementação do botão
}

export default Controles;
```

Usamos a importação padrão, pois neste caso tudo o que queremos expor do módulo está contido na classe `Controles`. Na classe `Player`, temos de importar os módulos, algo semelhante ao código seguinte:

```
import Controles from './Controles';
import Titulo from './Titulo';
import Links from './Links';

class Player {
    // implementação do player
}
```

17.4 RÓTULOS EM MÓDULOS

Não é necessário usar o nome exato das propriedades do módulo quando o importamos para dentro do nosso código; é possível utilizar rótulos. Muitas vezes os módulos que queremos usar possuem nomes pouco intuitivos, ou no caso mais comum, estão em uma língua estrangeira. Podemos usar rótulos para diferenciá-los. Por exemplo, considere a exportação da classe `Perfil`:

```
class Perfil {
  // implementação do Perfil
}

export default Perfil;
```

Se não quisemos usar o nome `Perfil` da classe no nosso projeto, podemos rotulá-la com outro nome por meio da palavra reservada `as`. Imagine que o módulo `Perfil` representa um jogador dentro de um projeto de um jogo. Como se trata de um jogo, podemos tratar `Perfil` como `Jogador`, por exemplo:

```
import Perfil as Jogador from './profile';
```

E o mesmo princípio pode ser aplicado para importações nomeadas.

```
import {adicionarPerfil as adicionarJogador} from ‘./Perfil’;

adicionarPerfil(); // ERRO!
adicionarJogador()// OK!
```

Só lembre de que, uma vez definido um rótulo, é obrigatório utilizá-lo dentro do módulo onde ele foi importado. Caso contrário, não funcionará.

17.5 ÚLTIMAS CONSIDERAÇÕES

Além do que já vimos, é importante ter em mente quando falamos de módulos que:

- **Módulos são *singletons*:** isso significa que mesmo que um módulo seja importado múltiplas vezes dentro de um projeto, somente uma "instância" dele vai existir.
- **Módulos podem importar coisas de outros módulos:** isso significa que é possível utilizar dentro de um módulo coisas que foram importadas de outros módulos que ele usa.
- **Importações de módulos são hoisted:** um detalhe bem importante que vale a pena citar é que tudo o que é importado é movido internamente para o topo do escopo atual. Em outras palavras, isso significa que o que você está importando está disponível em qualquer momento, mesmo que você tenha declarado o `import` do módulo só depois no código. Por exemplo:

```
somar(1,2); // 3
import {somar} from './matematica'
```

CAPÍTULO 18

FUNÇÕES GERADORAS

Funções geradoras (*generators*) são um dos novos recursos trazidos pelo ES6, mais complicado de se entender. Isso porque sua sintaxe é um pouco diferente do que estamos acostumados e do que vimos até então.

Além disso, à primeira vista, é uma funcionalidade que não parece fazer sentido. Mas não se preocupe, é uma sensação temporária. Veremos que funções geradoras são um dos recursos mais bacanas do ES6. Faremos um passo a passo com dificuldade gradual e, ao final deste capítulo, você estará dominando este recurso. Começaremos entendendo o que elas são.

18.1 O QUE SÃO FUNÇÕES GERADORAS?

Funções geradoras são funções no JavaScript em que podemos interromper e retornar sua execução várias vezes. Em termos práticos, isso significa que a execução do método é realizada até um ponto e é interrompida até que invocada novamente. Quando invocada, continua sua execução a partir do ponto em que parou.

Fazendo uma analogia, é mais ou menos como acontece com uma linha de ônibus. O ônibus continua percorrendo o seu itinerário até que aconteça algo — como a sinalização de um dos pedestres — que indique que ele deve parar. Uma vez que alguém sinalize que o ônibus pode partir, ele fecha as portas e continua o seu caminho até o final da linha, até que ele chegue lá, ou seja

interrompido em outro ponto.

Mas como será que isso funciona nas funções geradoras? Como a função sabe quando parar? E como ela sabe de onde continuar?

Para entendermos melhor o que isso quer dizer, tome como exemplo uma possível implementação para o nosso cenário do ônibus. Usaremos uma função geradora chamada `percorrerLinha477` que representa a trajetória do ônibus do terminal de partida até o fim de sua linha.

```
function* percorrerLinha477() {
    console.log('Passei pela rua 1');
    yield 'Fim da linha';
}
```

Repare atentamente na função geradora que acabamos de criar. Note que há duas coisas que nunca vimos antes: a palavra reservada `function` acompanhada de um `*` (asterisco), e a palavra reservada `yield`.

O asterisco é usado para rotular uma função como geradora. Isso indica ao interpretador JavaScript que essa função pode ter sua execução interrompida nos pontos definidos com a palavra `yield`. O asterisco pode ficar imediatamente depois da palavra reservada `function` (como fizemos anteriormente), ou imediatamente antes do nome da função. Ou seja, a função `percorrerLinha477` também poderia ser escrita assim:

```
function *percorrerLinha477() {
    console.log('Passei pela rua 1');
    yield 'Fim da linha';
}
```

A palavra reservada `yield`, como foi dito, é utilizada para definir um ponto de parada da função. A princípio, isso significa que, ao invocar a função, o código será executado, passando pelo `console.log` e imprimindo a mensagem `"Passei pela rua 1"`

no console, certo? Quase.

Ao invocarmos a função `percorrerLinha477`, temos uma surpresa, pois a mensagem `"Passei pela rua 1"` não é exibida:

```
function *percorrerLinha477() {
    console.log('Passei pela rua 1');
    yield 'Fim da linha';
}
percorrerLinha477(); // {}
```

Mas por que a mensagem não foi exibida? O `yield` não define um ponto de parada da função? Acontece que a mensagem não foi exibida porque, quando chamamos uma função geradora, seu corpo não é executado imediatamente. Em vez disso, um objeto iterável é retornado. Esse objeto possui uma função muito útil chamada `next`. Ao utilizar este método `next` dele, o corpo da função geradora é executado até a primeira expressão `yield`, que define o valor que será devolvido no retorno da função.

Resumindo, para que possamos ver a saída esperada, precisamos chamar a função geradora, e então invocar o método `next` no objeto retornado. Deste modo, o corpo do método é executado até encontrar a palavra reservada `yield`. Vejamos:

```
const linha = percorrerLinha477(); // {}
linha.next(); // Passei pela rua 1
```

Para ver o valor definido pelo `yield` retornado, vamos atribuir o valor de `iteravel.next` em uma variável e imprimir seu valor no console:

```
const linha = percorrerLinha477(); // {}
const parada = linha.next(); // Passei pela rua 1
console.log(parada); // { value: 'Fim da linha', done: false }
```

Nossa simulação até então pode ser representada neste diagrama:

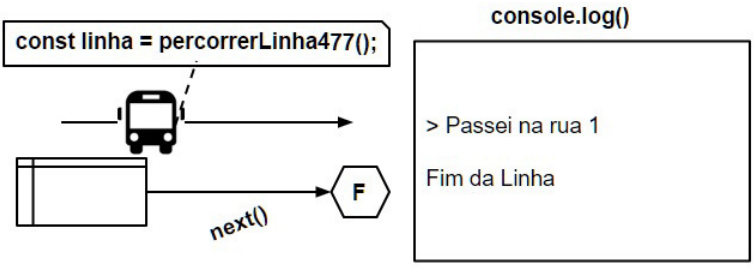


Figura 18.1: Linha do ônibus representada por uma função geradora

Note que, ao imprimir a variável `parada`, não recebemos somente a frase `"Fim da linha"` como uma `String`, mas sim um objeto literal com duas propriedades: `value` e `done`. A propriedade `value` representa o valor definido na declaração do `yield`, no nosso caso, a mensagem "Fim da linha". Já o `done` é responsável por nos dizer se todas as execuções daquela função geradora já foram feitas.

Como só temos a declaração de um `yield` no nosso método, ou seja, somente um ponto de parada, ao chamar o método `next` do objeto iterável retornado novamente, o `done` retorna o valor `true`.

```
const linha = percorrerLinha477(); // {}
linha.next(); // Passei pela rua 1
const parada = linha.next();
console.log(parada); // { value: undefined, done: true }
```

Assim como uma linha de ônibus pode ter várias paradas, uma função geradora também pode. Para isso, basta declarar vários pontos de parada dentro da função com o `yield`. A cada invocação do `next`, o código é executado entre os dois pontos de parada.

Agora pensando novamente na linha de ônibus, podemos implementar várias paradas, em que em cada uma delas ele imprime o nome da rua por qual ele acabou de passar, como no diagrama a

seguir:

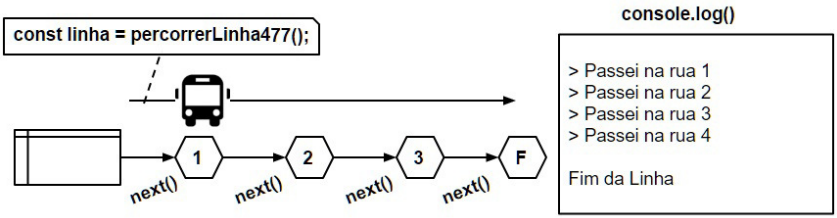


Figura 18.2: Linha do ônibus com múltiplos pontos de parada representada por uma função geradora

A implementação é relativamente simples:

```
function* percorrerLinha477() {
    console.log('Passei pela rua 1');
    yield 'Parada 1';
    console.log('Passei pela rua 2');
    yield 'Parada 2';
    console.log('Passei pela rua 3');
    yield 'Parada 3';
    console.log('Passei pela rua 4');
    yield 'Fim da linha';
}

const linha = percorrerLinha477();
let next = linha.next(); console.log(next);
next = linha.next(); console.log(next);
next = linha.next(); console.log(next);
next = linha.next(); console.log(next);
next = linha.next(); console.log(next);
```

Ao executar este código, temos:

```
Passei pela rua 1
{ value: 'Parada 1', done: false }
Passei pela rua 2
{ value: 'Parada 2', done: false }
Passei pela rua 3
{ value: 'Parada 3', done: false }
Passei pela rua 4
{ value: 'Fim da linha', done: false }
{ value: undefined, done: true }
```

18.2 ITERAÇÕES COM GERADORES

No último código que executamos no item anterior, tivemos de invocar repetitivamente o método `next`. Além de o código ficar extremamente repetitivo, foi necessário saber de antemão quantos valores a função geradora retornava para não invocar o `next` no objeto iterável desnecessariamente.

Para contornar isso, podemos tentar melhorar este código utilizando um laço de repetição do tipo `while` que verifica o valor da variável `next.done` a cada iteração. Vamos experimentar:

```
function* percorrerLinha477() {
    console.log('Passei pela rua 1');
    yield 'Parada 1';
    console.log('Passei pela rua 2');
    yield 'Parada 2';
    console.log('Passei pela rua 3');
    yield 'Parada 3';
    console.log('Passei pela rua 4');
    yield 'Fim da linha';
}

const linha = percorrerLinha477();
let next = linha.next();
while(!next.done) {
  console.log(next);
  next = linha.next();
}
console.log(next); // exibe o último
```

Ao executar este trecho de código, vemos que o resultado continua intacto:

```
Passei pela rua 1
{ value: 'Parada 1', done: false }
Passei pela rua 2
{ value: 'Parada 2', done: false }
Passei pela rua 3
{ value: 'Parada 3', done: false }
Passei pela rua 4
{ value: 'Fim da linha', done: false }
{ value: undefined, done: true }
```

Mas apesar de esta estratégia funcionar e de fato diminuir a repetição de código, ela ainda não está perfeita. Precisamos fazer um controle da propriedade `done` no nosso laço para saber quando parar a repetição.

Não seria bacana se houvesse alguma estrutura de repetição que já entendesse como iterar funções geradoras? Pois bem, existe e este é o grande segredo por trás das funções geradoras. Lembra de que alguns capítulos atrás estudamos o laço de repetição `for...of` ? Pois bem. Esta estrutura foi criada de modo a interpretar funções geradoras, e isso nos permite iterar qualquer tipo de estrutura de dados.

Como ela faz isso? Veremos logo mais. Por enquanto, vamos experimentar iterar nosso código nesta estrutura.

```
const linha = percorrerLinha477();
for(let parada of linha) {
  console.log(parada);
}
```

A execução deste código gera o mesmo resultado:

```
Passei pela rua 1
Parada 1
Passei pela rua 2
Parada 2
Passei pela rua 3
Parada 3
Passei pela rua 4
Fim da linha
```

Não é simplesmente fantástico como as peças se encaixaram?

18.3 ENTENDENDO O SYMBOL.ITERATOR

Para realmente entender como a estrutura `for...of` consegue interpretar as funções geradoras e, a partir delas, permitir iterar qualquer tipo de estrutura de dados, é preciso entender o conceito

do `Symbol` e do `Symbol.iterator`, outra novidade do ES6. O símbolo representa um tipo único e imutável de dado.

Se usarmos um símbolo como propriedade/índice de um objeto, ele é armazenado de uma forma especial em que a propriedade não vai aparecer em enumerações das propriedades de um objeto. Como no exemplo a seguir, no qual declaramos um `Symbol` com o nome `simbolo`, e então tentamos extrair seu valor.

```
const objeto = {
    valor: 10,
    [Symbol("simbolo")]: "Oi, sou um símbolo",
};

console.log(objeto.valor); // 10
console.log(objeto.simbolo); // undefined
```

No ES6, o `Symbol.iterator` especifica o iterador padrão de um objeto. Quando usamos a estrutura `for...of` para iterar um objeto, o método definido por este símbolo é chamado e o iterável retornado é usado para obter os valores a serem iterados. Isso significa que todos os tipos de objetos iteráveis por padrão no JavaScript possuem este método definido.

Vamos tomar como exemplo o objeto do tipo `Array`. Podemos recuperar sua propriedade de iteração e usar para iterá-lo:

```
const primos = [2,3,5];
const iterador = primos[Symbol.iterator]();

console.log(iterador.next()); // { value: 2, done: false }
console.log(iterador.next()); // { value: 3, done: false }
console.log(iterador.next()); // { value: 5, done: false }

console.log(iterador.next()); // { value: undefined, done: true }
```

No caso das funções geradoras, elas também possuem um método de iteração na propriedade `Symbol.iterator` definido. É por isso que, quando as colocamos no laço do tipo `for...of`, ele é iterado perfeitamente. O mesmo pode ser feito para qualquer

estrutura de dado que você definir. Por exemplo, vamos supor que temos uma estrutura que representa uma equipe de desenvolvimento de software de uma startup.

```
const equipe = {
  quantidade: 3,
  maturidade: 'alta',
  senior: 'Luís',
  pleno: 'Márcia',
  junior: 'Christian'
}
```

Como podemos fazer para iterar os integrantes desta equipe, sendo que as propriedades `quantidade` e `maturidade` não interessam? Podemos definir uma função geradora no `Symbol.Iterator` para a nossa estrutura, de modo que ela retorna somente os membros da equipe:

```
const equipe = {
  quantidade: 3,
  maturidade: 'alta',
  senior: 'Luís',
  pleno: 'Márcia',
  junior: 'Christian',
  [Symbol.iterator]: function* () {
    yield this.senior;
    yield this.pleno;
    yield this.junior;
  }
}
```

Agora, ao iterar a equipe no laço `for...of`, temos o resultado esperado:

```
for(let integrante of equipe) {
  console.log(integrante);
}

// Luís, Márcia, Christian
```

18.4 DELEGAÇÃO DE FUNÇÕES GERADORAS

Por fim, também podemos combinar duas funções geradoras

através da delegação de geradores. Imagine, por exemplo, que temos uma equipe de projetos que é subdividida em dois times: o time de desenvolvimento e o time de negócios. Vamos implementar este cenário.

Primeiro, criamos um objeto literal para cada time, sendo que cada um deve conter: a quantidade de envolvidos, seus respectivos cargos e uma função geradora para que possamos iterar estes times.

```
const timeDesenvolvimento = {
  quantidade: 3,
  senior: 'Luís',
  pleno: 'Márcia',
  junior: 'Christian',
  [Symbol.iterator]: function* () {
    yield this.senior;
    yield this.pleno;
    yield this.junior;
  }
}

const timeNegocios = {
  quantidade: 2,
  diretor: 'Marcelo',
  vice: 'Guilherme',
  [Symbol.iterator]: function* () {
    yield this.diretor;
    yield this.vice;
  }
}
```

Agora que temos nossos times, precisamos definir que nossa equipe de projetos é constituída das duas.

```
const equipe = {
  timeDesenvolvimento,
  timeNegocios
};
```

Para que seja possível criar uma função geradora que itere nossa equipe de projetos, assim como fizemos para os times de desenvolvimento e negócios, utilizamos a delegação de geradores. Usamos a palavra reservada `yield` acompanhada de um `*`

(asterisco). Ele funciona como um link, uma porta, entre as duas funções geradoras.

```
const equipe = {
  timeDesenvolvimento,
  timeNegocios,
  [Symbol.iterator]: function* () {
    yield* timeDesenvolvimento;
    yield* timeNegocios
  }
};
```

Agora ao iterar a nossa equipe com o `for...of`, temos o nome de todos os integrantes da nossa equipe.

```
for(let integrante of equipe) {
  console.log(integrante);
}

// Luís
// Márcia
// Christian
// Marcelo
// Guilherme
```

CAPÍTULO 19

OPERAÇÕES ASSÍNCRONAS COM PROMISES

Uma das grandes novidades do ES6 — e que já era aguardada há bastante tempo pela comunidade — é o suporte nativo a promises. De forma simples e resumida, promises são objetos que nos auxiliam a trabalhar com operações assíncronas. Este tipo de objeto aguarda a operação ser completada e oferece uma resposta positiva (resolvida) para quando realizada com sucesso, ou negativa caso algo tenha ocorrido algum erro no processo (rejeitada).

Promises já estão no ecossistema do JavaScript há muitos anos. Bibliotecas como o JQuery, Axios e Bluebird, já oferecem implementações de promises, cada um com sua particularidade. E elas funcionam muito bem. Mas com o ES6, os próprios navegadores darão suporte nativo a promises.

Mas para quem nunca trabalhou com promises ou não conhece exatamente o que este termo significa, faremos uma breve explicação do que ela é, qual seu propósito e, em seguida, já mostramos como utilizá-las com o ES6.

19.1 O QUE SÃO PROMISES?

Promises são uma alternativa criada para lidar com resultados

de operações assíncronas. Como a execução do JavaScript é ininterrupta (mesmo com o uso de funções como `setTimeout` ou `setInterval`), temos um problema quando um certo trecho do nosso código depende do resultado de uma operação que não sabemos quanto tempo demorará até ser completada.

Uma situação típica na qual temos este problema é quando consumimos APIs REST em nossas aplicações. As solicitações são feitas para um webservice e não sabemos de antemão quanto tempo este serviço vai demorar para enviar uma resposta. Ao mesmo tempo, o trecho de código seguinte à requisição necessita do resultado desta operação; caso contrário, não tem como executar sua lógica. Resultado: podemos ter o valor indefinido no momento que o código é executado.

Um exemplo é o consumo de uma API de um serviço de clima. Se tentarmos acessar o valor da temperatura antes que a requisição tenha sido concluída, não teremos o valor que queremos, como representado no diagrama:

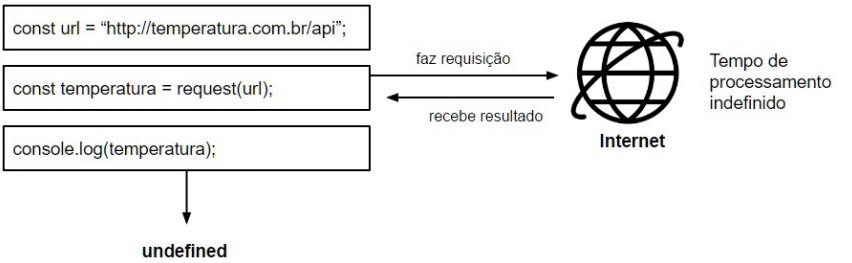


Figura 19.1: Simulação de requisição para uma API

A abordagem mais convencional para lidar com operações deste tipo são os `callbacks` . Para uma operação em que o tempo de execução é indefinido, usamos uma função de retorno que será chamada somente quando a operação for finalizada. Como no trecho a seguir, onde passamos dois argumentos para a

`funcaoAssincrona` , sendo um deles a função que é chamada no final de sua execução:

```
funcaoAssincrona(arg1, callback){
  // faz request e afins
  // e no final da execução executamos o callback
  callback();
}

function callback() {
  // operação que quero fazer depois que tiver a resposta da reque
st
}
```

O problema desta abordagem é que, uma vez que você começa a trabalhar com múltiplas operações assíncronas, fica complicado entender a sequência de operações que está sendo executada, pois cada trecho vai depender do outro, que depende do outro, assim sucessivamente. No final, você acabará com um código assim:

```
obj.funcaoAssincrona(function(response) {
  response.funcaoAssincrona(function(response2) {
    response2.funcaoAssincrona(function(response3) {
      response3.funcaoAssincrona(function(response4) {
        return response4;
      });
    });
  });
});
```

A manutenção e o debug de códigos assim se tornam uma tarefa extremamente complicada (e uma dor de cabeça enorme para os desenvolvedores). Este problema é mundialmente conhecido como *callback hell*.

O QUE É O CALLBACK HELL?

Este nome foi dado para a situação na qual temos várias chamadas assíncronas que dependem uma da outra. Como as operações assíncronas ocorrem simultaneamente e respondem em tempos diferentes, torna-se uma tarefa extremamente árdua para o desenvolver entender o que acontece na execução deste código.

Para mais informações, confira: http://callbackhell.com/.

Para não ter que lidar mais com este problema, surgiram as implementações das promises, que nada mais são do que objetos que contêm referências para funções que são executadas quando as operações assíncronas são terminadas com êxito ou falhas. Vamos ver como elas funcionam.

19.2 OS ESTADOS DAS PROMISES

Como vimos, uma promise guarda a promessa de que a função que a gerou vai, em algum momento, terminar e devolver uma resposta, podendo ser esta positiva ou negativa. Também vimos que a principal desvantagem de se utilizar `callbacks` é que acabamos criando cascatas de funções assíncronas, o que torna o código muito difícil de ler e debuggar.

Uma promise, em sua essência, possui três estados:

- **Não resolvido:** estado inicial, quando está esperando algo ser finalizado;
- **Resolvido:** estado no qual a operação foi concluída, sem erros;

- **Rejeitado:** estado no qual a operação foi concluída, porém, com erros.

Para os dois últimos estados, resolvido e rejeitado, associamos funções que queremos que sejam executadas. Para isso, usamos as palavras reservadas `then` e `catch`.

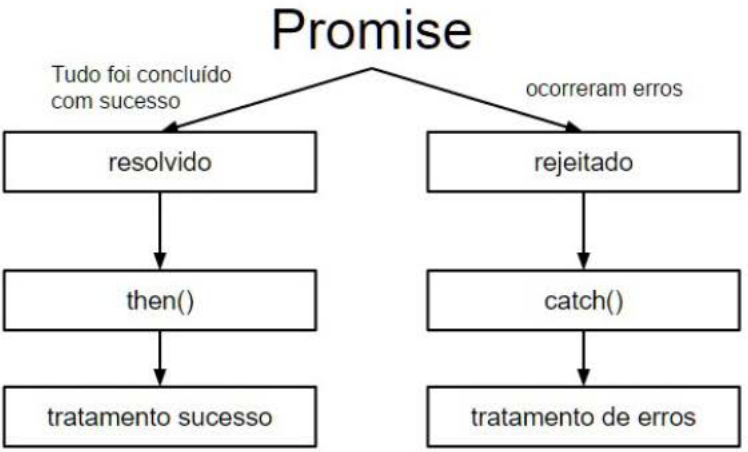


Figura 19.2: Estados de uma Promise

Vamos ver como estruturar uma promise e trabalhar com estes estados.

19.3 O ESQUELETO DE UMA PROMISE

Até o momento, vimos que promises possuem três estados: não resolvido, resolvido e rejeitado; e que os dois últimos possuem funções associadas que são definidas pelas palavras reservadas `then` e `catch`, respectivamente. Sabendo disso, já podemos começar a codificar e construir o esqueleto de uma promise.

```
let promise = new Promise((resolve, reject) => {
  // corpo da promise
});
```

```
promise.then();
promise.catch();
```

Por padrão, o construtor da promise recebe uma função com dois argumentos: `resolve` e `reject`. Utilizamos estes parâmetros dentro da lógica de nossa promise para indicar quando ela foi resolvida ou rejeitada. Quando um promise é resolvida sem problemas, o `then` é automaticamente ativado, assim como acontece com o `catch` quando a promise é rejeitada. Vamos ver um exemplo mais prático que simula esta situação com o código a seguir:

```
let promise = new Promise((resolve, reject) => {
  let resultado = true;
  if(resultado) {
    resolve("deu tudo certo!");
  } else {
    reject("deu tudo errado!");
  }
});
```

Com este código, criamos uma situação simulada do uso de uma promise. A variável `resultado` contém o que seria o resultado da operação assíncrona. Se tudo deu certo, o `resolve` é invocado com a `String` "deu tudo certo". No caso de falha, utilizamos o `reject` com a `String` "deu tudo errado". Para recuperar estes dados que estão sendo passados, usamos o parâmetro `data` que por padrão também é definido dentro das funções `then` e `reject`.

```
promise.then((data) => console.log(`resultado positivo: ${data}`))
;
promise.catch((data) => console.log(`resultado negativo: ${data}`)
);
```

Agora, ao executar nosso código, teremos o resultado positivo:

```
resultado positivo: deu tudo certo!
```

Ao alterar o valor da variável `resultado` para `false`, ao executar o código, o resultado final é diferente:

```
resultado negativo: deu tudo errado!
```

19.4 OPERAÇÕES ASSÍNCRONAS

Para facilmente simular a execução de uma função assíncrona dentro da nossa promise — como uma chamada AJAX, por exemplo — vamos utilizar a função `setTimeout`.

```
let promise = new Promise((resolve, reject) => {
  let resultado = false;
  let tempo = 2000; // milisegundos
  setTimeout(() => {
    if(resultado) {
      resolve("deu tudo certo!");
    } else {
      reject("deu tudo errado!");
    }
  }, tempo)
});

promise.then((data) => console.log(`resultado positivo: ${data}`))
;
promise.catch((data) => console.log(`resultado negativo: ${data}`)
);
```

Colocamos a execução do nosso código dentro do `setTimeout` e definimos o tempo em uma variável com o mesmo nome, com o valor de 2000, que representam dois segundos. Para tornar o cenário ainda mais semelhante ao do mundo real, vamos executar algum código fora da promise. Vamos colocar a seguinte linha no final do código:

```
console.log('fui executado antes!');
```

Agora, ao rodar o nosso código. Repare no resultado:

```
fui executado antes!
resultado negativo: deu tudo errado!
```

Repare que a mensagem `"fui executado antes!"` surgiu no console quase que instantaneamente, mesmo sendo a última linha na ordem de execução do código. Dois segundos depois, a segunda

linha foi exibida, uma vez que a promise foi rejeitada.

Este experimento simula exatamente o que acontece quando lidamos com operações assíncronas. Ai que vemos o verdadeiro poder das promises: permitir lidar com operações assíncronas de uma maneira muito semelhante à maneira síncrona.

19.5 ANINHAMENTO DE THEN E CATCH

As funções `then` e `catch` possuem várias características importantes que precisamos saber antes de sair usando. A primeira delas é que podemos aninhar suas chamadas. Por exemplo, na declaração de uma promise, em vez de declarar os métodos `then` e `catch` de modo separado, como fizemos antes, podemos fazer de modo aninhado:

```
promise
  .then((data) => console.log(`resultado positivo: ${data}`))
  .catch((data) => console.log(`resultado negativo: ${data}`));
```

Do mesmo modo, podemos aninhar várias chamadas do `then`. Isso faz com que cada um seja executado após o outro:

```
promise
  .then((data) => console.log(`resultado positivo: ${data}`))
  .then((data) => console.log(`resultado positivo 2: ${data}`))
  .catch((data) => console.log(`resultado negativo: ${data}`));
```

Entretanto, é preciso estar atento quando usamos aninhamento de `then`. Repare que, ao executar o código do exemplo anterior, com a promise resolvida, notaremos algo estranho na saída do console:

```
resultado positivo: deu tudo certo!
resultado positivo 2: undefined
```

Por que a segunda linha contém um `undefined`? Não deveria ser um "deu tudo certo!" como na primeira linha?

Isso acontece porque o valor da variável `data`, disponível na primeira chamada do `then`, não é passada adiante. Ou seja, o valor da variável `data` é sempre correspondente ao retorno da função anterior. Isso significa que, para que tenhamos a mensagem "deu tudo certo!" nas duas execuções do `then`, precisamos alterar nosso código. Retornaremos a variável `data` dentro da execução do `then`:

```
promise
  .then((data) => {
    console.log(`resultado positivo: ${data}`);
    return data;
  })
  .then((data) => console.log(`resultado positivo 2: ${data}`))
  .catch((data) => console.log(`resultado negativo: ${data}`));
```

Agora sim, ao executar o código, temos as duas mensagens sendo exibidas, como esperado:

```
resultado positivo: deu tudo certo!
resultado positivo 2: deu tudo certo!
```

19.6 COMO LIDAR COM ERROS INESPERADOS

Quando utilizamos o `reject` nas nossas promises, estamos tratando os casos em que já esperamos de antemão que algum cenário possa dar problema. Entretanto, há situações nas quais erros que o desenvolvedor não esperava podem acontecem. No exemplo a seguir, vamos inserir um erro intencionalmente na execução do código para simular este cenário.

```
let promise = new Promise((resolve, reject) => {
  throw new Error('erro!');
  resolve('ok!');
});

promise
  .then((data) => console.log(`sucesso: ${data}`))
  .catch((data) => console.log(`falha: ${data}`));
```

Mesmo esperando que o `resolve` ative a função `then`, como um erro está sendo lançado, o `catch` é ativado.

```
falha: Error: erro!
```

O mesmo acontece quando temos vários `then` aninhados. Se um deles dá algum erro, o `catch` é automaticamente ativado. No exemplo seguinte, jogamos um erro propositalmente na primeira execução do `then`:

```
let promise = new Promise((resolve, reject) => {
  resolve('ok!');
});

promise
  .then((data) => {
   console.log(`sucesso: ${data}`);

   throw new Error("erro!");

   return data;
  })
  .then((data) => console.log(`sucesso: ${data}`))
  .catch((data) => console.log(`falha: ${data}`));
```

Ao rodar este código, vemos que o segundo log de sucesso nunca é exibido.

```
sucesso: ok!
falha: Error: erro!
```

CAPÍTULO 20

METAPROGRAMAÇÃO COM PROXIES

O último recurso do ES6 que estudaremos neste livro são os proxies. Estes objetos nos permitem interceptar e customizar operações fundamentais em objetos, tais como:

- Acessar uma propriedade;
- Setar uma propriedade (nova ou não);
- Enumerar propriedades;
- Deletar propriedades;
- Verificar a existência de uma da propriedade.

Um proxy, como veremos detalhadamente a seguir, é um objeto que representa um outro objeto. Ele possui a incrível capacidade de interceptar chamadas às propriedades do objeto que ele representa, tendo o poder de alterar o resultado dela.

Os proxies fazem parte do que chamamos de metaprogramação. Vamos entender um pouco melhor o que isso significa antes de nos aprofundar nos seus poderes.

20.1 O QUE É METAPROGRAMAÇÃO?

Como muitas tarefas são extremamente repetitivas na programação, surge a necessidade de fazer a mesma coisa de novo, de novo e de novo. Para contornar isso, criamos algoritmos que

automatizam estas tarefas.

Entretanto, nem sempre é possível eliminar toda a repetição. Quando não conseguimos criar diretamente no programa algo que as evite, escrevemos um programa diferente que altera o primeiro de forma automatizada. Este processo é o que chamamos de metaprogramação.

Talvez a maneira mais fácil e objetiva de defini-la seja como: a programação de programas que manipulam a si mesmo e/ou a outros programas. Um metaprograma é todo programa que atua sobre ele mesmo ou outro programa, seja no formato fonte, binário, ou em uma representação abstrata em memória. Em resumo: um programa que gera programas.

Fazendo uma analogia bem simples, é como ter uma fábrica que cria fábricas que criam carros. A fábrica de carros cria o carro que é o produto final, mas utilizamos uma outra fábrica para criar as fábricas que geram os produtos finais.

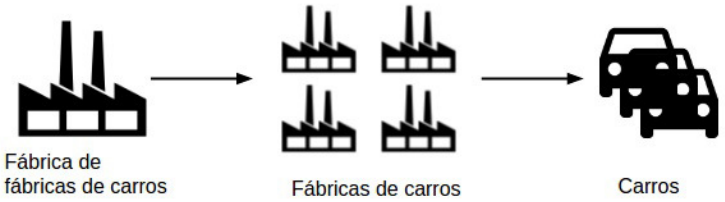


Figura 20.1: Analagia para metaprogramação — Uma fábrica que cria fábricas

Existe uma outra analogia bem bacana em um artigo chamado *An Introduction to Metaprogramming*, escrita por Ariel Ortiz em 1997 e publicada no portal Linux Jornal (http://www.linuxjournal.com/article/9604), que está mais próximo da realidade e nos ajuda a entender o que significa falar em um programa que gera programas. Vou fazer uma adaptação da

situação hipotética que foi construída no artigo (não se preocupe em entender os códigos descritos, mas sim o que eles significam).

Gerando programas com programas

Imagine que Erika é uma estudante muito inteligente do curso de graduação de ciência da computação em uma faculdade de renome. Mesmo sem formação acadêmica, Erika já trabalhou muitos anos na área e possui bastante experiência nas linguagens de programação C e Ruby.

Um certo dia, na aula de introdução a programação, o professor do curso, o Dr. José Luiz Gomez, a pegou conversando com o namorado dela em um chat pela internet. Como punição, ele obrigou que Erika escrevesse um programa em C que exibisse 1.000 linhas da seguinte frase: "Eu não vou conversar em sala de aula". Mas não pense que o desafio era fácil, havia duas restrições que ela deveria seguir na implementação:

- O programa não poderia ter nenhum tipo de laço de repetição (`for` , `do...while` , ...);
- O programa não poderia conter nenhum tipo de instrução `goto` .

Ao dar esta punição, ele esperava que Erika fizesse algo do gênero:

```
#include <stdio.h>
int main(void) {
    printf("1) Eu não vou conversar em sala de aula.\n");

   // 998 printfs omitidos

    printf("999) Eu não vou conversar em sala de aula.\n");
    printf("1000) Eu não vou conversar em sala de aula.\n");

    return 0;
}
```

Como o professor era ingênuo, ele realmente achou que Erika fosse escrever a instrução `printf` 1000 vezes. Entretanto, ele não contava com a astúcia de Erika: ela encontrou uma solução para este problema na metaprogramação. Ela pensou: "Por que escrever esse programa extremamente repetitivo na mão se eu posso criar um simples programa que cria esse programa para mim?". Então, ela utilizou seus conhecimentos na linguagem Ruby para criar o seguinte script:

```
File.open('punishment.c', 'w') do |output|
    output.puts '#include <stdio.h>'
      output.puts 'int main(void) {'
      1.upto(1000) do |i|
          output.puts "    printf(\"#{i}. " +
          "Eu não vou conversar em aula.\\n\");"
      end
    output.puts '    return 0;'
    output.puts '}'
end
```

O script Ruby gera um arquivo chamado `punishment.c` que cria o programa em C que gera os mil `printf` que o professor solicitou. Ela mostrou esse programa ao professor e ele não teve outra escolha se não aceitar e parabenizá-la pelo trabalho. A metaprogramação salvou o dia.

É por isso que classificamos proxies como uma funcionalidade de metaprogramação. Eles não atuam no nível da aplicação, mas atuam no nível que atuam na aplicação. Eles conseguem fazer com que o programa gere outros programas, como vimos no exemplo, ou processe a si mesmo e o altere através de introspecção, automodificação e intersecção. Hora de ver como fazer isso.

20.2 VOLTANDO PARA OS PROXIES

Agora que entendemos melhor o conceito de metraprogramação, podemos entender como este conceito se aplica

nos proxies. Dissemos que um proxy é um objeto que representa um outro objeto. Vamos ver o que isso significa na prática.

Imagine que estamos na etapa de implementação de um sistema e estamos tendo certas dificuldades para trabalhar no desenvolvimento de uma funcionalidade. O programa está dando problemas em certa etapa e não sabemos exatamente o porquê. Para facilitar o nosso debug, vamos criar um mecanismo de log nos objetos. A principal classe da operação é a `Usuario` , representada a seguir:

```
class Usuario {
    constructor(login, senha) {
        this.login = login;
        this.senha = senha;
    }
}
```

Para começar a validação, criamos um usuário e verificamos se ele está funcionando:

```
const usuario = new Usuario('SuperJS', '123');
console.log(usuario.login); // SuperJS
console.log(usuario.senha); // 123
```

A classe parece estar funcionando bem, mas como não temos certeza de como ela se comporta nos algoritmos, vamos utilizar um proxy para nos ajudar. Para todas as propriedades que forem chamadas dentro do objeto, queremos que saia no console um log nos avisando que a propriedade foi acessada. Para fazer isso, podemos usar um proxy que intercepta o acesso às propriedades do objeto.

Em outras palavras, utilizaremos um proxy para alterar o comportamento do objeto. Podemos criar isso com o código a seguir:

```
const proxy = new Proxy(usuario, {
  get(alvo, propriedade) {
    console.log(`${propriedade} foi solicitada!`);
```

```
    return alvo[propriedade];
  }
});
```

Vamos analisar passo a passo o que foi feito nesse código. Em primeiro lugar, instanciamos um objeto do tipo `Proxy`. O construtor de um proxy aceita dois argumentos: o primeiro é o `alvo`, o objeto que queremos representar, ou seja, o objeto que vamos interceptar as chamadas; o segundo é um objeto chamado por convenção de *handler*. Este define o comportamento do proxy.

As chaves do objeto handler são chamadas de *traps* (armadilhas) com os valores sendo funções que definem como o proxy se comportará quando essa trap for disparada. Anteriormente, usamos apenas a trap de `get`, que é disparada quando tentamos ler uma propriedade do `alvo`. Podemos validar nosso proxy invocando as propriedades do `usuario`.

```
console.log(proxy.login);
// login foi solicitada!
// SuperJS

console.log(proxy.senha);
// senha foi solicitada!
// 123
```

Repare que, para cada chamada da propriedade do `usuario`, tivemos duas saídas. Isso porque antes do console imprimir o valor da propriedade, o proxy interceptou essa chamada e jogou o que havíamos definido no proxy.

Neste caso, usamos somente a trap para o `get`, mas existe uma série de traps disponíveis que podemos utilizar:

- `getPrototypeOf`
- `setPrototypeOf`
- `isExtensible`
- `getOwnPropertyDescriptor`
- `defineProperty`

- `has`
- `set`
- `deleteProperty`
- `ownKeys`
- `apply`
- `construct`

Vamos ver mais um caso de uso para proxies.

20.3 VALIDAÇÕES DE INPUTS COM PROXIES E TRAPS

Podemos usar proxies para fazer validações de inputs. Tomando novamente o nosso `Usuario` como exemplo, imagine que agora temos uma nova propriedade na classe: `idade`. Nosso objetivo agora é ter certeza de que o usuário vai preencher a idade com um número. Qualquer outra entrada deve dar erro.

Para realizar isto, podemos novamente usar o proxy, mas desta vez vamos utilizar a trap de `set`. Esta aceita três parâmetros: o `alvo`, a `propriedade` e o `valor`.

```
const validador = {
    set(alvo, propriedade, valor) {
        if(propriedade === 'idade') {
            if (!Number.isInteger(valor)){
                throw new TypeError('A idade não é um número!');
            }
        }

        alvo[propriedade] = valor;
    }
}
```

Agora que criamos nosso `validador`, podemos testá-lo. Primeiro, vamos colocar um valor válido.

```
const usuario = new Proxy({}, validator);
usuario.idade = 10;
```

```
console.log(usuario.idade); // 10
```

Agora vamos tentar valores que não sejam números e ver se o código gera os erros:

```
const usuario = new Proxy({}, validator);
usuario.idade = 'dez'; // TypeError: A idade não é um número!
usuario.idade = {}; // TypeError: A idade não é um número!
usuario.idade = true; // TypeError: A idade não é um número!
```

Realmente bem poderoso, né? Mas como já dizia um velho sábio dos quadrinhos: "Com grandes poderes, vem grandes responsabilidades". Como os proxies alteram o comportamento dos objetos, é preciso ser bem cauteloso no seu uso; caso contrário, pode causar grandes problemas e inconsistências nos sistemas. Podemos acidentalmente acabar alterando comportamentos de objetos que não planejávamos e podemos levar um bom tempo até descobrir.

20.4 DESATIVANDO UM PROXY

É possível revocar um proxy, ou seja, desligá-lo de certa forma. Para isso, precisamos construir nosso proxy com um `Proxy.revocable`. Ele nos retornará o `proxy` e o método `revoke`. Note que não é necessário utilizar a palavra reservada `new` para criar um proxy revocável.

```
const {proxy, revoke} = Proxy.revocable(alvo, handler);
```

Feito isso, podemos agora invocar o método `revoke`. Este pode ser chamado várias vezes, entretanto, somente na primeira chamada ele terá efeito e desativará o proxy. Uma vez feito isso, toda operação realizada no proxy resultará em um `TypeError`, como no trecho a seguir:

```
const {proxy, revoke} = Proxy.revocable({}, {});
proxy.teste = 'teste';

console.log(proxy.teste); // teste
```

```
revoke();
console.log(proxy.teste)
// TypeError: Cannot perform 'get' on a proxy that has been revoke
```

E por fim, mais um detalhe importante: proxies desativados não podem mais ser ativados!

20.5 ÚLTIMAS CONSIDERAÇÕES

Metaprogramação é uma poderosa arma para os desenvolvedores. Ela nos permite alterar o comportamento dos objetos e customizá-los para resolver uma série de problemas e situações. No entanto, é necessário sempre estar atento à performance. Tome sempre cuidado com as alterações e, sempre que possível, faça medições dos impactos.

Capítulo 21

UM FUTURO BRILHANTE PARA O JAVASCRIPT

O ES6 trouxe para a linguagem JavaScript o que ela precisava para finalmente ser levada a sério como tecnologia. As mudanças não foram poucas, muito pelo contrário. Como não tínhamos mudanças relevantes desde a ES5.1, foi necessário praticamente reaprender a linguagem para incorporar todos os seus novos conceitos e melhorias.

A especificação permitiu que a linguagem se tornasse mais próxima das demais linguagens focadas em Orientação a Objetos, como o Java e o C#, por exemplo. Por debaixo dos panos, tudo continua como antes, mas temos uma linguagem mais flexível, limpa, objetiva e convidativa para novos programadores.

Ao longo deste livro, passamos ponto a ponto pelas principais mudanças propostas pela especificação. Explicamos os conceitos, vimos suas aplicações em contextos da vida real, e praticamos bastante com os exercícios. O conhecimento adquirido não foi pouco. Você deve se sentir orgulhoso.

Vimos como iterar listas com os protocolos de iteradores e iteráveis, o laço de repetição `for...of` e os métodos auxiliares, como: `forEach`, `find`, `reduce` e `map`. Entendemos as novas estruturas de dados do JavaScript, o `Map`, `WeakMap`, `Set` e `WeakSet`, quais suas particularidades, por que foram criadas e

quando utilizá-las em nosso código.

Aprendemos a diferença entre a declaração de variáveis com o `var` e os novos `const` e `let`. Observamos como os templates literais tornaram a manipulação de strings muito mais fácil. Notamos como as arrow functions tornaram a implementação de funções mais enxuta e eficiente, e como as melhorias nos objetos literais tornaram a manipulação de objetos melhor e o código mais limpo.

Compreendemos como atribuir valores padrões para parâmetros sem gambiarras, implementações muito semelhantes ou controles de fluxo (`if`). Aprendemos como utilizar os operadores `rest` e `spread` para tratar parâmetros nas nossas funções. Percebemos como o desestruturamento de objetos nos permite interagir com os objetos de forma descomplicada, sem a necessidade de acessar elementos que não são do interesse, como costuma acontecer quando estamos consumindo um webservice.

Com classes e módulos, praticamos como tornar o nosso código muito mais organizado, estruturado e, principalmente, modularizado, tornando possível o reaproveitamento. Também desenvolvemos as funções geradoras, um novo recurso poderoso para interagir e iterar não somente listas, mas objetos de qualquer estrutura de dados. Com os proxies, averiguamos como podemos lidar com operações síncronas e assíncronas das nossas aplicações na internet de forma inteligente e segura. E por fim, mas não menos importante, estudamos um recurso avançadíssimo de metaprogramação com os proxies, que nos permite modificar os objetos da maneira que quisermos.

Ufa! Quanta coisa!

Realmente espero que este livro tenha contribuído com o seu aprendizado dos recursos ES6. Esse conhecimento sem dúvidas será

um diferencial para o seu futuro como desenvolvedor de software.

Ficou alguma dúvida? Encontrou algo estranho? Precisa de uma ajuda para entender os conceitos? Não tenha receio de entrar em contato nos canais oficiais e/ou nas redes sociais. Acredito que, somente ajudando um ao outro, a comunidade de desenvolvedores brasileira vai se fortalecer e se superar.

Entre em contato!

http://www.diegopinho.com.br

Mas se você acha que o aprendizado termina aqui, você está muito enganado. Com o engajamento da comunidade de desenvolvedores, o envolvimento de grandes empresas de tecnologia e a promessa de uma nova especificação por ano, o JavaScript continuará evoluindo. Isso significa que novas funcionalidades, melhorias e correções estão por vir, e a base de todas elas será o ES6.

O ES7, ou ES2016, já está batendo na nossa porta. Precisamos estar preparados, porque ainda há uma longa estrada pela frente.

Agradeço sua atenção e confiança. Muito obrigado!

Até a próxima e bons códigos!

Não deixe de acompanhar o site oficial para novidades

http://www.entendendoes6.com.br