FERNANDO FELTRIN

JavaScript DO ZERO

À PROGRAMAÇÃO ORIENTADA A OBJETOS

1ª EDIÇÃO
2021

JS

CAPA

FERNANDO FELTRIN

JavaScript DO ZERO

À PROGRAMAÇÃO ORIENTADA A OBJETOS

1ª EDIÇÃO
2021

JS

JavaScript do ZERO à Programação Orientada a Objetos

Fernando Feltrin

SOBRE O AUTOR

Fernando Feltrin é Engenheiro da Computação com especializações na área de ciência de dados e inteligência artificial, Professor licenciado para docência de nível técnico e superior, Autor de mais de 10 livros sobre programação de computadores e responsável por desenvolvimento e implementação de ferramentas voltadas a modelos de redes neurais artificiais aplicadas à radiologia (diagnóstico por imagem).

AVISOS

LIVROS

Disponível em [Amazon.com]

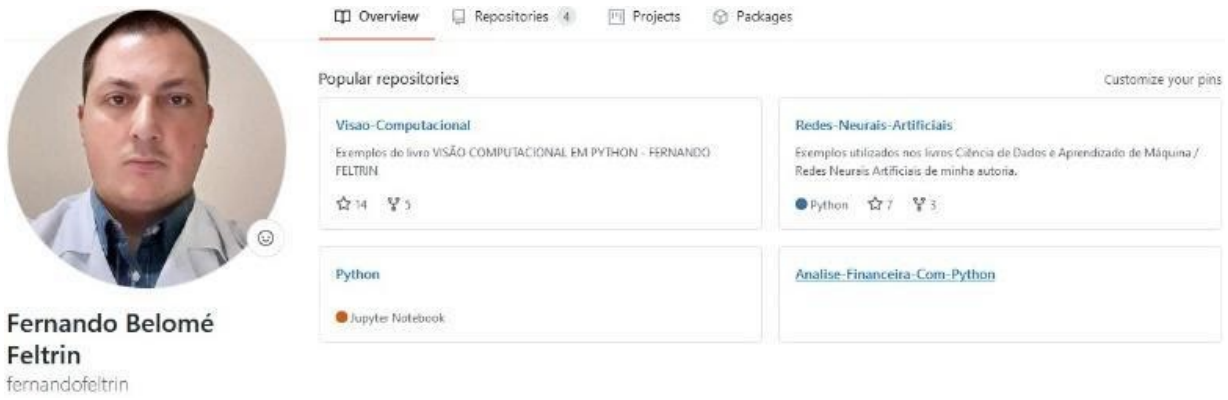
CURSO

Curso Python do ZERO à Programação Orientada a Objetos

Mais de 15 horas de vídeos objetivos que lhe ensinarão programação em linguagem Python de forma simples, prática e objetiva.

REDES SOCIAIS

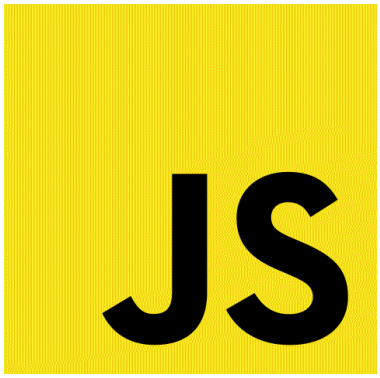
https://www.facebook.com/fernandofeltrinpython



https://github.com/fernandofeltrin

ÍNDICE

JAVASCRIPT



Por quê programar? E por quê em JavaScript?

Dos inúmeros nichos da chamada Tecnologia da Informação, especificamente falando de toda parte de lógica computacional, certamente uma das áreas mais importantes é a que se dedica a criar meios e métodos para programação de computadores.

Se tratando deste nicho específico, dentro do mesmo existe todo um universo de possibilidades no que diz respeito a forma como escrevemos programas para que por meio destes tenhamos ferramentas para os mais diversos problemas computacionais que podem serem resolvidos de forma lógica.

O perfil de profissional de TI moderno, que o mercado de trabalho busca para atender suas necessidades, é justamente o profissional com uma bagagem de conhecimento híbrida, no sentido de que o mesmo tenha conhecimento de aspectos físicos (hardware) e lógicos (software) de modo que saber programação para assim criar suas próprias ferramentas é um enorme diferencial para este tipo de profissional.

Programar em si, por mais abstrato que o conceito possa parecer, nos dias atuais é indispensável, haja visto que o crescimento exponencial de todas as áreas de tecnologia demanda profissionais que de alguma forma, em algum nível, tenha conhecimento de como criar soluções por meio de aplicações, tudo isto a nível de programação.

Nessa linha de raciocínio, somada a formação base do profissional de TI, a programação de computadores se faz um diferencial muito positivo para sua carreira. No âmbito acadêmico inclusive, podemos notar que ao longo das décadas muitos dos cursos foram se adaptando as necessidades do mercado incluindo em suas grades curriculares uma carga horária dedicada a ensinar ao menos as bases de alguma linguagem de programação.

Se tratando de programação em si, vivemos um momento muito privilegiado no que diz respeito ao vasto número de linguagens de

programação assim como no que se refere as ferramentas que temos em mãos para escrever nossos programas.

Diferentemente de outras épocas onde era preciso passar instruções para um interpretador quase a nível de linguagem de máquina, hoje temos linguagens de alto nível onde por meio das mesmas, uma vez entendida sua sintaxe e regras básicas, podemos criar aplicações bastante robustas de forma fácil e intuitiva, voltando nosso foco em si totalmente à programação e não a forma de se programar.

Com uma rápida pesquisa em algumas fontes certamente você irá se deparar com nomes de linguagens de programação como C, Java, Python e JavaScript (entre tantas outras) como principais linguagens em uso pela comunidade de desenvolvedores em geral. O fato é que muitas das linguagens já consolidadas evoluíram muito com o passar das décadas assim como novas linguagens foram surgindo, aumentando ainda mais o leque de possibilidades para os desenvolvedores.

Quando estamos falando de linguagens de programação, cada uma terá um propósito, seja este geral ou específico, mas o importante a salientar é que na prática não existe de fato uma linguagem melhor que outra, mas sim a que melhor se adapta dentro dos critérios do desenvolvedor e ao tipo de problema computacional a ser resolvido.

JavaScript é uma dessas linguagens de programação em destaque, pois é uma linguagem moderna, de fácil curva de aprendizado e que oferece ferramentas para programação em todas as áreas possíveis, desde um simples programa com funções internas e interface com o usuário até aplicações web ou em computação distribuída, aplicativos mobile, internet das coisas via sistemas embarcados a até mesmo modelos de visão computacional e redes neurais artificiais por meio bibliotecas que implementam tais funcionalidades ao núcleo da linguagem, fazendo desta uma das melhores linguagens para se programar em todas as camadas de desenvolvimento de uma determinada aplicação.

O ponto é que JavaScript pode ser sua linguagem de programação definitiva, que vá atender a todos seus requisitos, de modo que a mesma dificilmente oferecerá alguma limitação ou restrição, muito menos se tornará defasada ao longo dos anos graças a comunidade que a usa e a mentem. Logo, independentemente se você já é programador de outra linguagem ou está entrando neste mundo do absoluto zero, tenha em mente que a linguagem de programação que você aprenderá a partir dos capítulos subsequentes deste livro é uma excelente ferramenta de programação, que possibilitará uma abordagem em todos os possíveis mecanismos de uma aplicação.

Por fim, espero que o conteúdo deste pequeno livro seja de grande valia para seu aprendizado dentro desta área incrível.

Metodologia

Este pequeno material foi elaborado com uma metodologia autodidata, totalmente prática, de modo que cada tópico e cada conceito será explicado de forma progressiva, sucinta e exemplificada.

Cada tópico terá seu referencial teórico seguido de um exemplo onde cada linha de código será explicada em detalhes. Quando necessário será feita uma “engenharia reversa” do código explicando os porquês de cada argumento e elemento dentro do código.

Desde já tenha em mente que aprender a programar requer atenção e muita prática, sendo assim, fortemente recomendo que sempre que possível pratique replicando os códigos dos exemplos assim como na medida do possível tente criar seus próprios códigos a partir da base entendida no exemplo em questão.

O conteúdo deste livro é um compendio introdutório sobre programação de computadores fazendo uso da linguagem JavaScript, todo conteúdo incluso no mesmo visa criar bases sólidas para que você consiga posteriormente trabalhar com ferramentas baseadas em JavaScript, logo, todos os códigos apresentados serão exemplos rodando em back-end, explorando as funcionalidades da linguagem que podem ser aplicadas a qualquer contexto e aplicação.

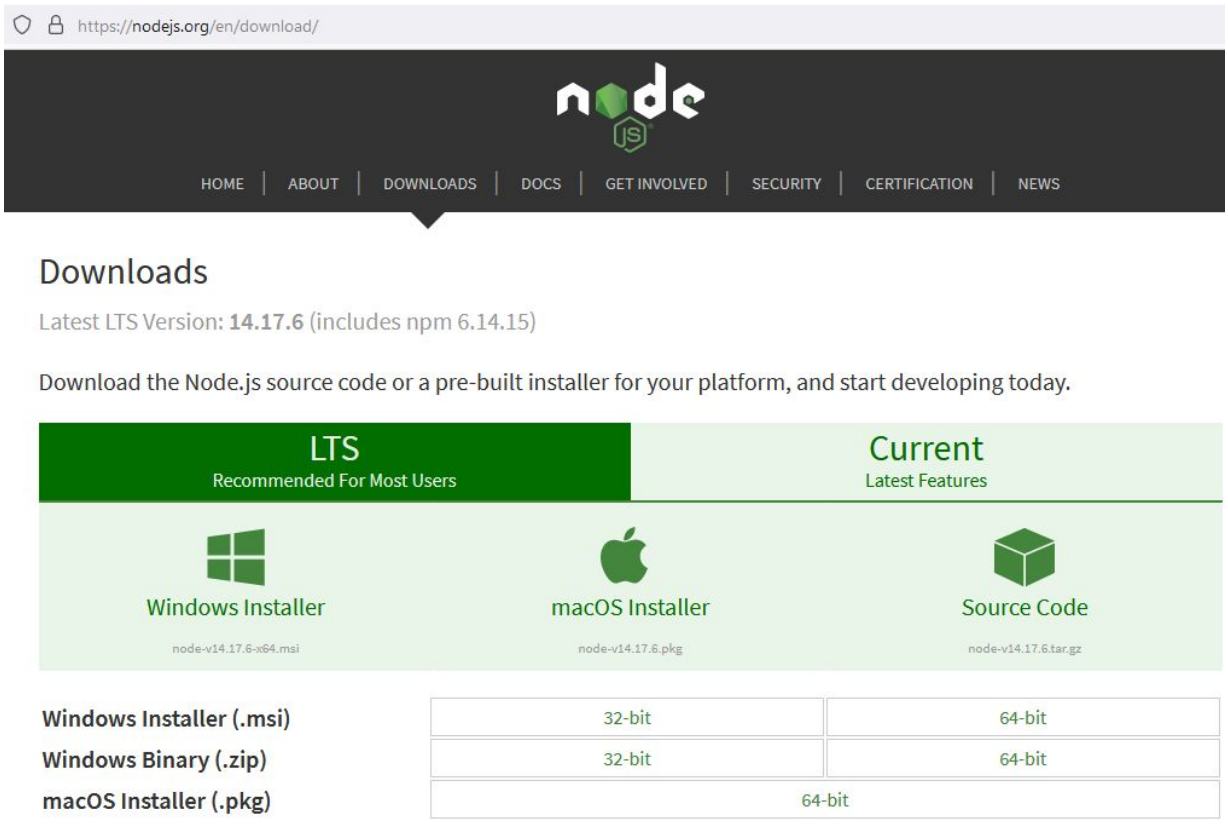
Finalizadas as considerações iniciais, vamos para a prática!!!

PREPARAÇÃO DO AMBIENTE

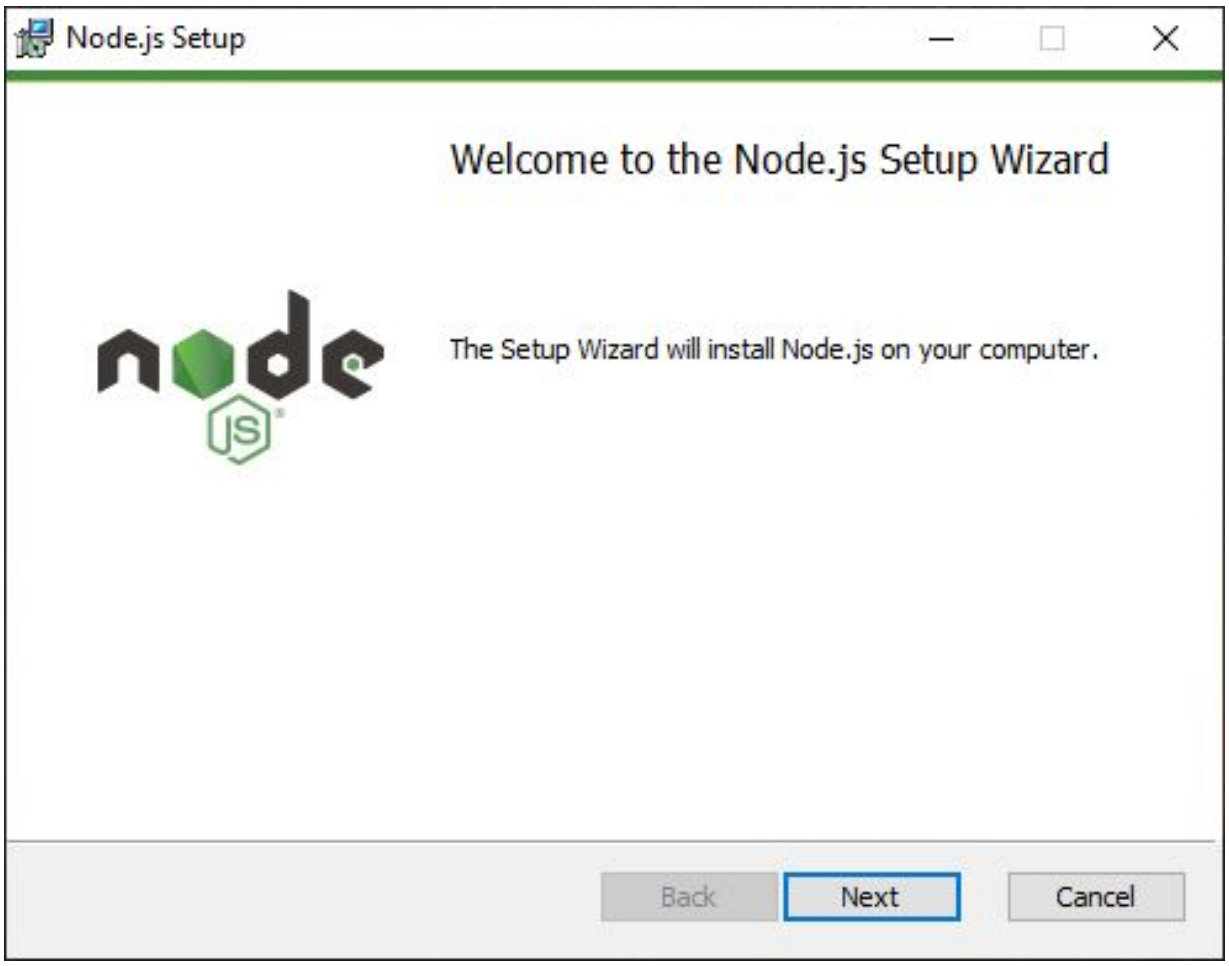
O primeiro passo a ser dado no que diz respeito à configuração de nosso ambiente de desenvolvimento é realizar as devidas instalações das ferramentas as quais faremos uso.

Se tratando de JavaScript é importante salientar que suas ferramentas de desenvolvimento não vêm nativamente instaladas em nosso sistema operacional, logo, para que possamos de fato dar início a nosso aprendizado, o qual faremos de forma totalmente prática, temos de realizar algumas simples instalações e configurações em nosso sistema.

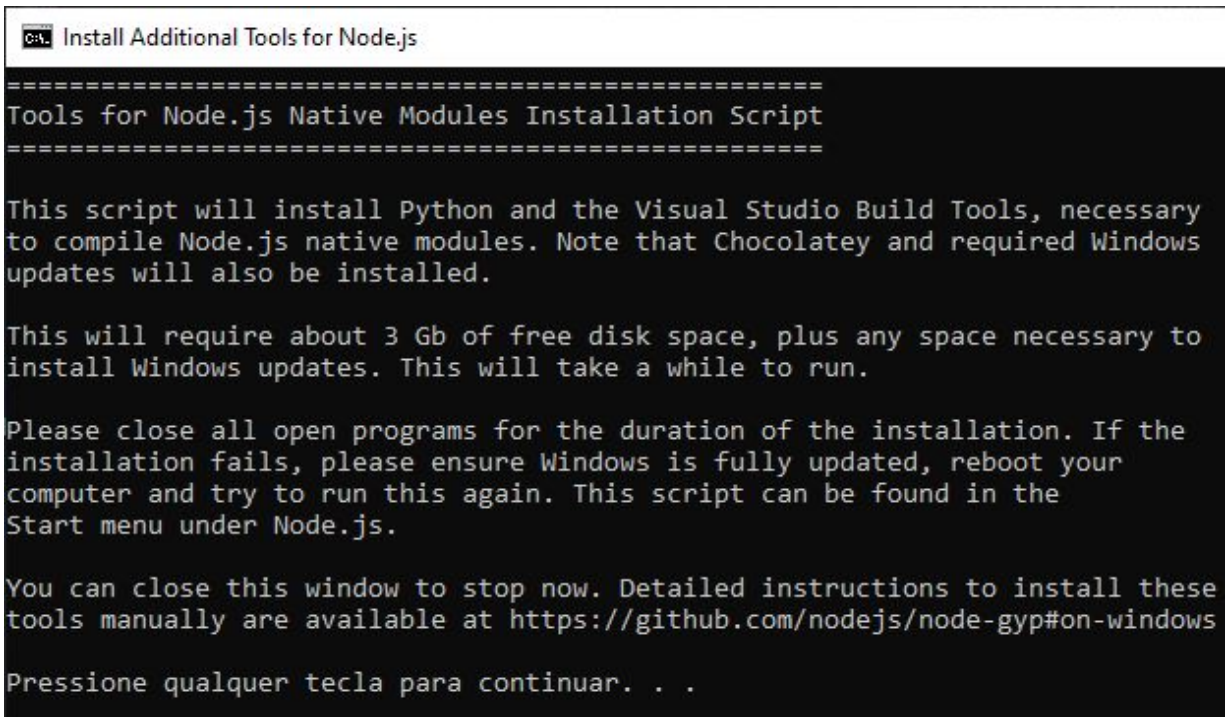
O processo de instalação do núcleo JavaScript pode ser feito de diversas formas, eu recomendo fortemente que tal instalação seja feita através da ferramenta Node.js, que veremos em detalhes em capítulos subsequentes, pois por meio desta ferramenta o processo de instalação de todas as dependências será totalmente automatizado.



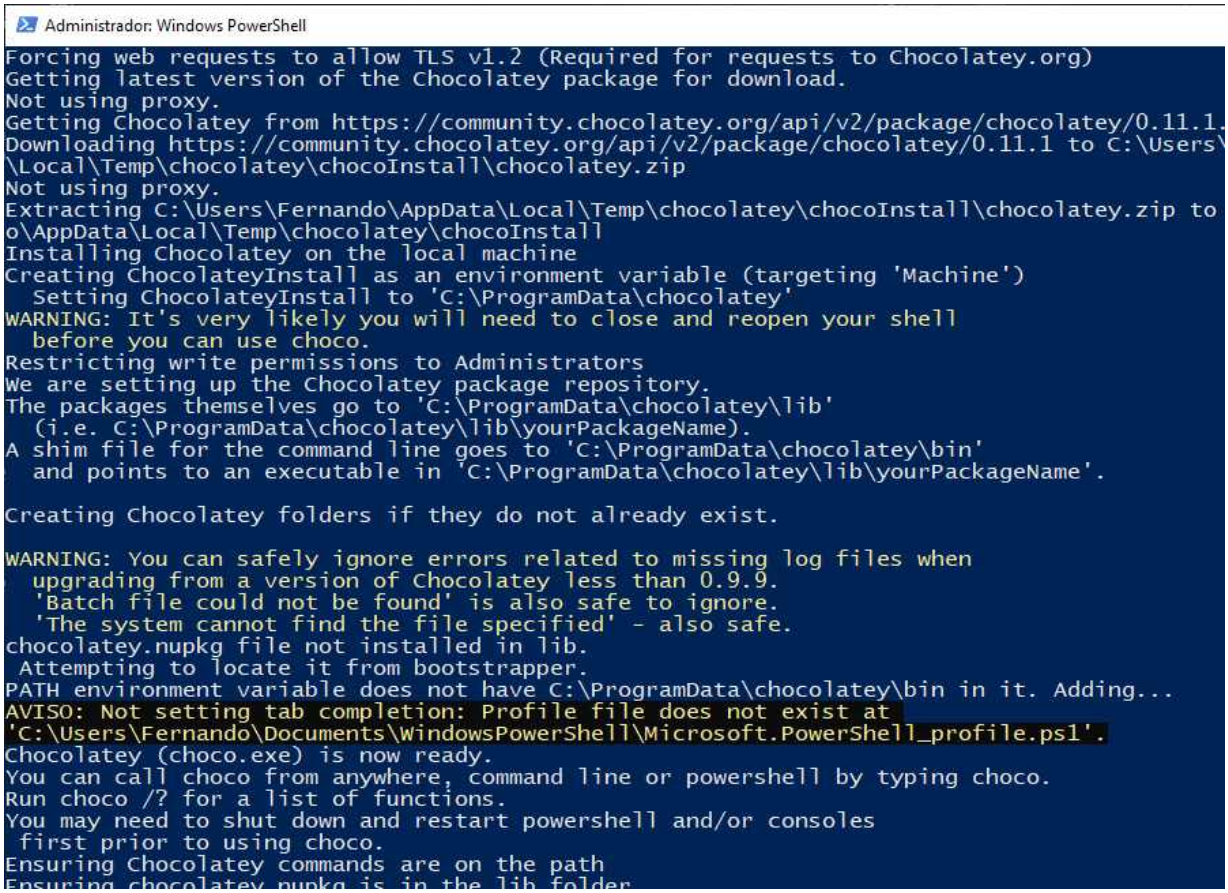
Acessando o site oficial da ferramenta Node.js, é possível baixar, independentemente de seu sistema operacional, tanto a última versão estável da ferramenta quanto versões experimentais. Apenas para fins de exemplo, todos os códigos apresentados neste livro estarão rodando sobre a versão LTS 14.17.6 do Node.js. Talvez versões diferentes possam apresentar alguma incompatibilidade, alteração ou descontinuidade ao tentar replicar os códigos apresentados neste livro.



O processo de instalação se dá como o de qualquer outro programa, através de uma interface de instalação a qual apenas avançamos algumas etapas aceitando a licença de uso, caminho de instalação, dentre outras configurações básicas.



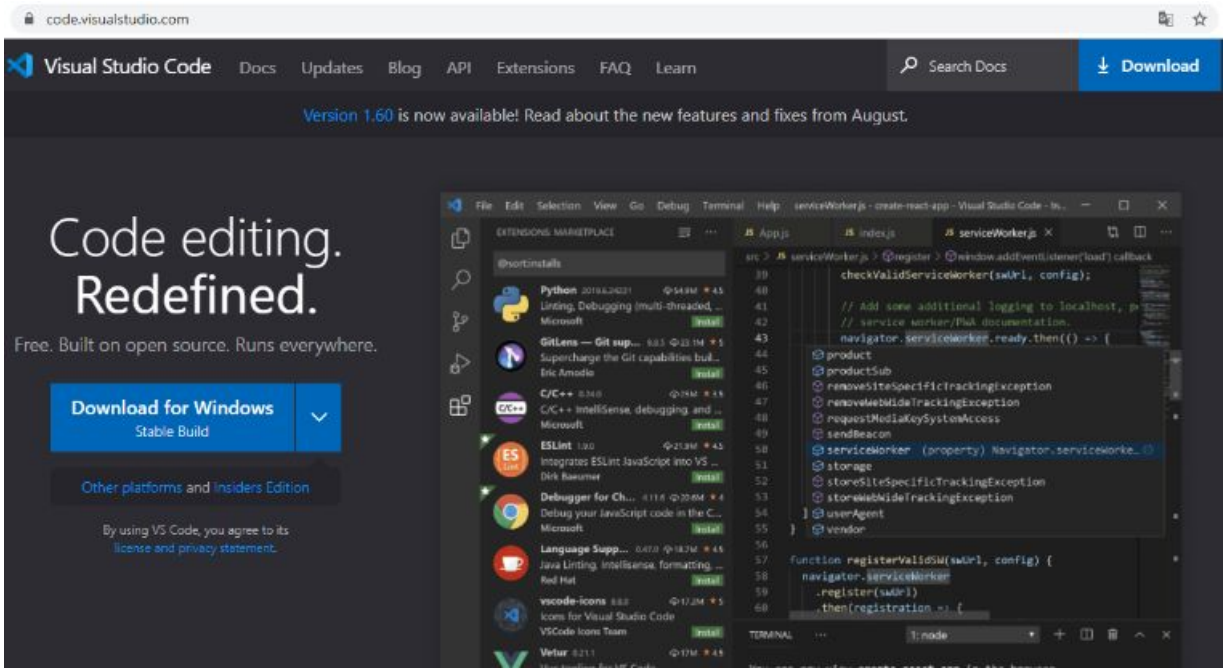
Não havendo nenhum erro na primeira etapa de instalação, será exibida uma tela de prompt de comando pedindo permissão para que se instale todas as dependências necessárias para pleno uso da linguagem JavaScript.



Concedidas as devidas permissões, é iniciado um script para instalação de todas as ferramentas adicionais.

Uma vez terminadas as devidas instalações das dependências, precisamos instalar uma IDE, outro tipo de ferramenta, dedicada ao processo de oferecer uma plataforma onde podemos não somente escrever nossos códigos como executar os mesmos.

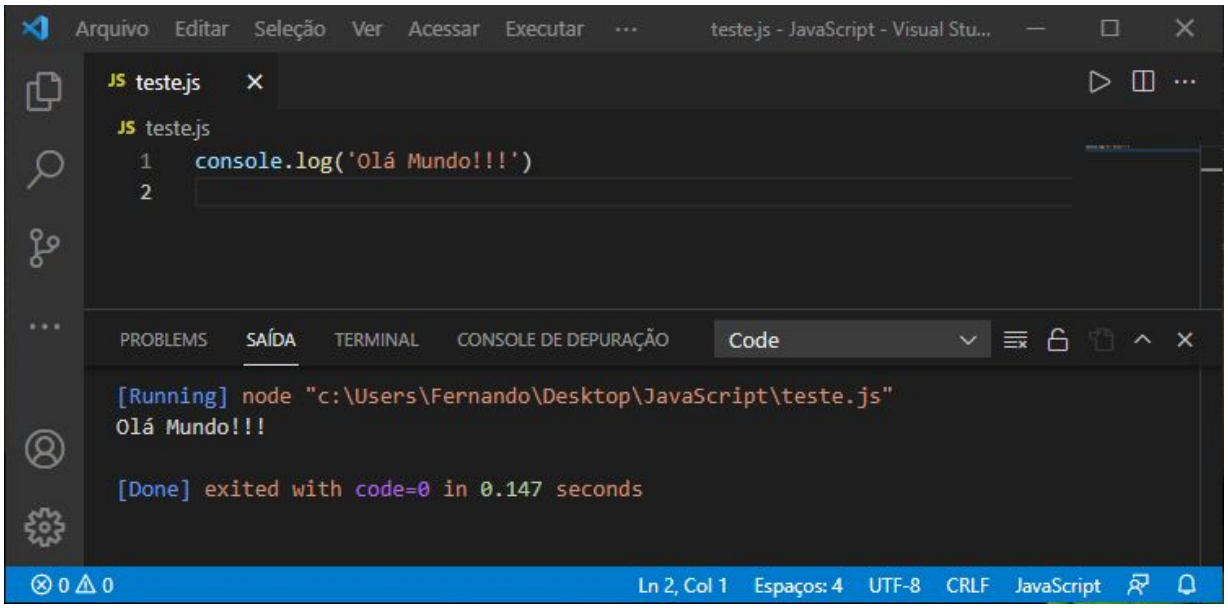
Existe uma vasta gama de IDEs disponíveis no mercado, cada uma com suas particularidades. Em nossos exemplos estaremos usando do Visual Studio Code, um editor de texto com recursos de IDE, de licença opensource, que nos fornecerá todas as ferramentas necessárias para nosso processo de codificação.



Para baixar a última versão estável do Visual Studio Code basta acessar seu site oficial e realizar o download do pacote de instalação. Assim como para o Node.js, o processo de instalação do Visual Studio Code é realizado por meio de um instalador próprio, semelhante ao de qualquer programa em ambiente Windows.

Terminadas as instalações, ao abrir o Visual Studio Code é necessário realizar a instalação de um plugin chamado Code Runner, que como nome sugere, assim como sua descrição, é uma extensão que implementa a funcionalidade de poder executar nossos códigos diretamente a partir do Visual Studio Code, sem a necessidade de nenhuma outra ferramenta adicional.

Se tratando da linguagem JavaScript, existem diversas formas de executar nossos códigos, sendo as principais a partir de um browser, para fins de visualizar o comportamento do código rodando na camada de interface com o usuário, ou a forma que usaremos neste livro, rodando códigos diretamente via IDE, recebendo os retornos da execução dos mesmos através do próprio terminal da IDE.



Uma vez realizadas corretamente todas as etapas anteriores, finalmente podemos partir para a prática pois temos em mãos todo o necessário para escrever e executar nossos códigos.

Apenas a nível de exemplo, haja visto que veremos em detalhes por parte do código da imagem acima em capítulos subsequentes, na imagem é possível ver a interface a qual usaremos para criar nossos códigos. A partir da tela inicial de nossa IDE é criado um arquivo de nome testes.js, onde no corpo do mesmo escrevemos a linha de código console.log('Olá Mundo!!!').

Executando nosso código é aberto no próprio editor uma aba com um terminal com o retorno esperado para esse código, que por sua vez simplesmente exibe em tela / terminal a mensagem 'Olá Mundo!!!'.

Neste momento não iremos nos adentrar aos detalhes de como este código é internamente interpretado e executado via Node.js, o ponto aqui é que agora temos um ambiente de desenvolvimento pronto para uso, logo, podemos finalmente dar início ao nosso aprendizado da linguagem JavaScript.

ESTRUTURA BÁSICA DE UM PROGRAMA

Quando estamos a escrever o código de um programa, basicamente estamos a abstrair uma ideia ou um problema computacional a ser resolvido em forma de texto, texto esse que nada mais é do que um script de comandos a serem interpretados pelo núcleo da linguagem, de modo que o mesmo realize certas funções conforme as instruções do código são lidas.

Dentro dessa linha de raciocínio, todo e qualquer programa tende a ser, salvo algumas exceções, escrito linha por linha, sequencialmente, de modo que o interpretador da linguagem irá ler e interpretar tais blocos de código executando suas devidas instruções.

Existem diversos tipos de programas, cada um com suas particularidades no que diz respeito ao modo de execução de seus códigos, mas uma característica comum a todos é o de processamento de algoritmos para um determinado propósito final, onde uma vez cumprido tal propósito, a finalidade do programa foi cumprida com êxito, podendo ser encerrado ou entrar em espera até que uma nova ação seja acionada.

Sintaxe Básica

Dentro desse contexto iniciado no tópico anterior, a primeira característica a ser levada em consideração em uma linguagem de programação é sua sintaxe. Como dito anteriormente, um código é escrito para ser interpretado, realizando certas funções de acordo com instruções recebidas pelo mesmo.

O ponto é que, independentemente de que linguagem de programação estivermos falando, cada uma delas vai possuir uma sintaxe própria, nomenclatura que usamos para nos referir a forma gramatical que devemos utilizar para que “a máquina” entenda nossas instruções.

Exatamente como quando aprendemos uma nova linguagem natural, passando por sua simbologia, gramática, regras e lógica de construção de argumentos. Quando estamos a aprender uma linguagem de programação estamos a aprender a forma de nos comunicarmos com um computador, de modo que o mesmo consiga entender comandos e executar tarefas a partir dos mesmos. Neste sentido, haja visto que uma “máquina” não possui nenhuma capacidade de abstração, todo código escrito deve estritamente respeitar a sintaxe esperada pelo interpretador da linguagem.

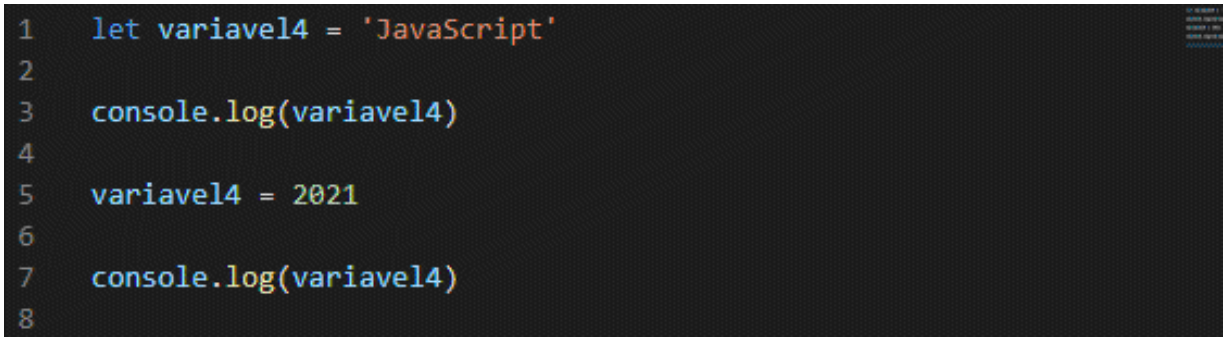
Logo nos capítulos subsequentes, à medida que estivermos aprendendo na prática os conceitos básicos da linguagem JavaScript, sempre que necessário estarei realizando as devidas explicações sobre tudo o que está envolvido no processo, a começar da sintaxe aplicada para a linguagem.

Por hora, devemos entender três conceitos básicos para que o processo de escrever nossos primeiros códigos se dê de forma correta.

Leitura Léxica

Primeiro conceito dentro da linha de raciocínio iniciada no parágrafo anterior é o que em programação chamamos de leitura léxica.

Alguns capítulos para trás, no processo de configurar um ambiente de desenvolvimento lhe foi sugerido a instalação de uma IDE, um editor de texto com ferramentas para interpretação de linguagens de programação, e nesse contexto é importante entendermos como um programa desses lê nossas linhas e blocos de código interpretando e executando as instruções contidas nos mesmos.



```
1  let variavel4 = 'JavaScript'
2
3  console.log(variavel4)
4
5  variavel4 = 2021
6
7  console.log(variavel4)
8
```

Apenas para fins de exemplo, observe o código acima. *Cada um dos elementos será explicado em detalhes logo nos próximos capítulos, por hora, vamos buscar entender a forma de leitura de um código por parte do núcleo da linguagem, a partir de uma IDE.

Quando estivermos a escrever nossos códigos, sempre estaremos abstraindo uma ideia para uma estrutura textual, que será escrita linha após linha, sequencialmente assim como em qualquer texto onde as ideias são descritas em uma sequência lógica dos fatos.

O interpretador da linguagem também realiza uma leitura em cada linha, da esquerda para a direita, lendo todos os elementos que constarem na mesma. Em determinadas situações, o interpretador irá integrar a leitura de múltiplas linhas, formando um bloco, para que assim assimile uma determinada instrução.

Em nosso exemplo, logo na primeira linha é criada / declarada uma variável de nome variavel4 que recebe como dado / valor

“JavaScript”, uma vez feita a leitura desta linha, uma série de mecanismos internos são acionados para que se crie esta estrutura lógica para nosso programa, assim como um espaço é alocado em memória para deixar os dados associados a esta variável carregados prontos para uso.

Em seguida, é lida a linha 2 que não possui nenhum código, sendo automaticamente ignorada, passando para a linha 3 que tem uma função que irá exibir em tela ao usuário o conteúdo atribuído a variável variavel4.

Em outras palavras, no momento que esta linha é lida, o interpretador vê que tipo de função está sendo chamada assim como seus parâmetros. Nesse caso, uma função para exibir algo em tela, seguido do parâmetro que faz referência a uma variável que terá seu conteúdo exibido em tela. O mesmo processo é repetido para as linhas 5 e 7 do código, assim como para todas as linhas subsequentes onde houverem sentenças de código.

O ponto que deve ficar claro aqui é que, o bloco de código em questão sendo executado, é gerado um retorno específico, porque o mesmo foi escrito de modo com que o interpretador conseguiu entender plenamente suas instruções, executando as mesmas. Isto se dá porque escrevemos em cada uma das linhas as devidas instruções de acordo com a sintaxe da linguagem, possibilitando assim criar, conforme o exemplo, um simples programa que quando executado irá exibir em tela JavaScript e 2021.

Caso houvesse qualquer erro de escrita, ou da ordem lógica como tais elementos foram escritos, o interpretador não iria conseguir entender as devidas instruções, parando a execução do programa imediatamente ou em situações mais brandas gerando comportamentos inesperados para o programa.

Em resumo, por parte de leitura léxica devemos sempre ter em mente que devemos escrever nossos códigos estritamente de acordo com a sintaxe da linguagem (exatamente de acordo com a sintaxe, pois o interpretador não possui nenhum artificio de

abstração para conseguir entender o que você “tentou escrever”), inserindo as devidas estruturas de código linha após linha dentro de uma sequência lógica dos fatos, para que dessa forma as instruções sejam de fato lidas, entendidas e executadas pelo interpretador.

Palavras reservadas

Complementar ao tópico anterior existe o que comumente chamamos de palavras reservadas ao sistema. Como veremos logo nos próximos capítulos, cada elemento a ser inserido em um código tem uma forma específica a qual o mesmo deve ser escrito para que seja identificado pelo interpretador.

Dentro dessa mesma lógica, raciocine que existem algumas palavras que não podemos usar como nome para nossas estruturas de código, pois são palavras que identificam ou acionam gatilhos de estruturas internas da linguagem com suas respectivas funcionalidades.

Por exemplo, no bloco de código do exemplo anterior existia logo na primeira linha, como primeiro elemento da sentença, a palavra “let”, que é um marcador / identificador interno para que o interpretador reconheça que ali, naquele ponto, está sendo declarada uma variável.

Logo, é simplesmente impossível usar “let” como nome de uma variável, pois let é uma palavra reservada ao sistema.

Apenas como exemplo, a seguir segue uma tabela com as principais palavras reservadas do núcleo da linguagem JavaScript.

break	Case	Catch	continue	default
delete	Do	Else	false	finally
for	Function	If	in	instanceof
new	Null	Return	switch	this
throw	True	Try	typeof	var
void	While	With		
abstract	boolean	byte	char	class
const	debugger	double	enum	export
extends	final	float	goto	implements

import	int	Interface	long	native
package	private	Protected	public	short
static	super	synchronized	throws	transient
volatile				

Indentação

Por fim, um último conceito básico essencial para o entendimento de nossos primeiros códigos é o de indentação. Assim como dito anteriormente que no processo de leitura léxica o interpretador lê linha por linha, sequencialmente, também é preciso entender que a tabulação de um código é reconhecida e identificada pelo interpretador, além é claro de tornar o código visualmente muito mais organizado. Embora em JavaScript não seja uma exigência a indentação, é uma boa prática de programação manter o código legível e organizado.

Certas estruturas de código, escritas em blocos de código, podem ter linhas de código associadas, linhas de código que devem ser lidas em conjunto, e o que identifica isso ao interpretador é, além da sintaxe de cada elemento logicamente, a forma como tais linhas e blocos de código são escritas, mais especificamente no que diz respeito a sua indentação / tabulação.

```
1   let inventario = {
2       camisa: 79.90,
3       tenis: 149.90,
4       meia: 9.90
5   }
6
7   console.log(inventario)
8
```

Novamente apenas para fins de exemplo, note que no bloco de código acima existem algumas particularidades. Primeira delas é que temos uma única estrutura de código escrita ao longo de 5 linhas diferentes, onde dentro dessa estrutura podemos ver sentenças (linhas de código) com espaços bem definidos e vírgula ao final das mesmas.

Dessa forma, o interpretador ao ler a sentença iniciada na primeira linha, ao ler a abertura de chaves, composta por três linhas de código com espaçamento e vírgula definidos, entende que tais

linhas de código fazem parte do corpo da estrutura iniciada na primeira linha.

Posteriormente veremos que por parte de sintaxe existem diversas estruturas de dados onde seu conteúdo pode se estender por diversas linhas, e isto é possível graças a forma como JavaScript em sua sintaxe usa de certos símbolos e espaçamentos para estruturar seus blocos de código de forma legível.

TIPOS DE DADOS

Partindo para a prática, uma das primeiras bases de conhecimento que precisamos ter é que em um código, independentemente da linguagem de programação utilizada, teremos dados categorizados em diferentes tipos, e essa diferenciação se faz necessária para que o interpretador da linguagem consiga, de acordo com o tipo de dado identificado, aplicar certas funções sobre o mesmo.

Em outras palavras, ao longo de nosso código teremos sentenças / expressões escritas em linhas onde estaremos usando de tipos de dados específicos para realização de determinadas tarefas, cada um dos mesmos com suas particularidades sintáticas.

Sendo assim, inicialmente temos de entender quais são os tipos de dados básicos da linguagem de programação, assim como sua sintaxe / forma de escrita, para então entendermos o comportamento de tal dado quando o posto à prova, sendo usado por alguma outra estrutura ou para uma determinada função.

Comentários

Antes mesmo de adentrarmos nos tipos de dados e suas particularidades, é interessante dedicarmos um pequeno tópico para entender os chamados comentários de um código.

Desculpe a redundância, mas um comentário serve para... comentar um código! Dentro desse contexto, entenda que uma prática comum é ao longo de nossos códigos, que dependendo da aplicação podem ser um tanto quanto extensos, inserirmos comentários para deixar alguma anotação a ser lembrada para aquela linha ou bloco de código.

Um comentário por sua vez possui uma sintaxe própria, e como sintaxe sempre devemos associar que sintaxe é a forma de escrita que identificará tal dado para o interpretador, também separando o mesmo dos demais, que nesse caso irá simplesmente ignorar tal comentário, conforme é esperado.

É considerada uma boa prática de programação realizar, quando necessário, comentários em um código para tornar o mesmo legível e explicativo para terceiros.

Um último ponto a se considerar para um comentário é que o mesmo não afeta em absolutamente nada o processamento de um código, pois como dito anteriormente, tal estrutura terá um identificador que irá sinalizar ao interpretador que tal linha ou bloco comentado não possui nada a ser executado.

```
1   // Comentário simples de até uma linha de extensão
2
```

Pela sintaxe JavaScript, um comentário simples é escrito com o prefixo “//”, de modo que após este marcador é perfeitamente possível escrever um comentário, desde que o mesmo possua apenas uma única linha de extensão.

Dependendo da IDE a qual você estiver usando, notará que cada tipo de dado terá uma cor diferente para que se diferencie

facilmente uns tipos de outros.

No caso de comentários, como em nosso exemplo, a coloração verde oliva nos identifica visualmente que ali existe uma linha de comentário.

```
/*
Comentário com mais de uma linha de extensão, inclusive
sem limites de linhas..
*/
```

Para situações onde se faz necessário um comentário mais longo, com múltiplas linhas de texto, usamos da notação de “/*” para abrir o campo de comentário e “*/” para encerrar este campo.

Dessa forma, podemos criar comentários um pouco mais elaborados em função de tal formato não possuir restrições.

```
//Comentário com mais de uma linha de extensão, inclusive
//sem limites de linhas..

```

É perfeitamente possível comentar linha por linha de um comentário usando de //, porém, esteticamente tal comentário fica com aparência de algo não funcional.

```
/*
* Padrão de comentário comumente usado em documentações,
* apenas por conveniência, pois em termos de funcionalidade
* é exatamente igual a qualquer outro comentário de várias
* linhas de extensão.
*/
```

Inclusive esteticamente falando, é muito comum encontrar em códigos de terceiros ou em documentações o formato de /* */ usando também de * a cada início de linha, apenas para manter a organização do comentário esteticamente mais agradável. Novamente, lembre-se que todo e qualquer comentário, independentemente do formato, não afeta a performance de execução de um código.

```
1   console.log('Fernando')
2   // A linha de código acima chama a função console.log()
3
```

```
Fernando
```

Sendo assim, devemos sempre prezar por um código limpo e funcional, evitando inserir comentários desnecessários.

Em nosso exemplo, logo após a primeira linha de código foi inserido um comentário simples explicando o que tal linha de código realiza.

```
1   console.log('Fernando')
2   /* A linha de código acima chama a função console.log(),
3   por sua vez parametrizada com uma string 'Fernando', que
4   como esperado será exibida em tela quando a linha de có-
5   digo referente à função for executada.
6   */
```

```
Fernando
```

Quando necessário, é possível a inserção de um comentário mais detalhado, composto de múltiplas linhas, desde que se respeite sua notação sintática.

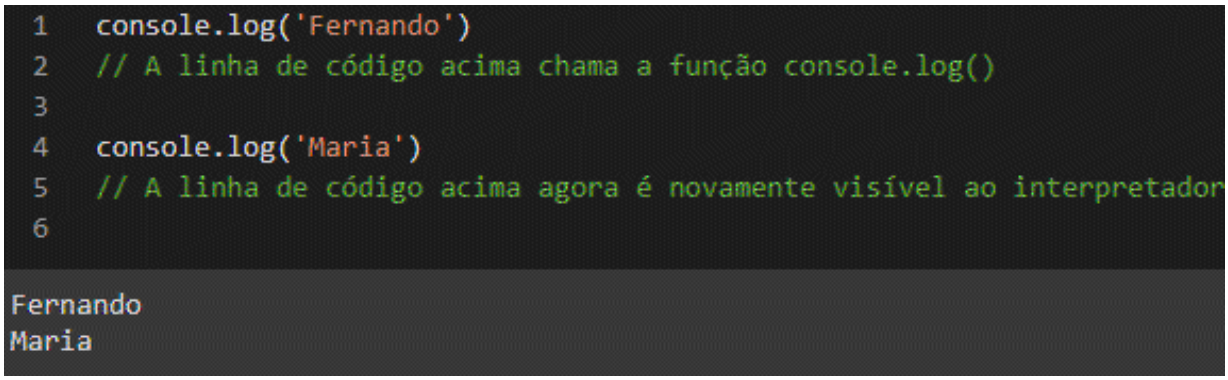
```
1   console.log('Fernando')
2   // A linha de código acima chama a função console.log()
3
4   //console.log('Maria')
5   // A linha de código acima está desabilitada
6
```

```
Fernando
```

Uma prática comum para certos contextos que veremos mais à frente será desabilitar para testes algumas estruturas de nosso código.

Como mencionado anteriormente, um comentário é algo que graças ao seu identificador fará com que o interpretador simplesmente ignore tal linha de código, logo, em nossa linha 4 do código,

inserindo // como prefixo, tornamos a função console.log( ) desta linha invisível ao interpretador pois “comentamos” a mesma.



“Descomentando” nossa linha 4 do código, a função console.log( ) para a estar visível e legível por parte do interpretador.

Nesse caso, após executada a função escrita na primeira linha, o interpretador irá ignorar o comentário inserido na linha 2, irá pular a linha vazia 4, e executar a função escrita na linha 4, que como esperado, exibirá em tela o nome ‘Maria’.

OBJETOS / VARIÁVEIS

Para finalmente darmos início ao entendimento dos tipos de dados básicos da linguagem JavaScript, devemos começar a entender alguns conceitos básicos os quais estaremos, quando for a hora certa, aprofundando ao longo deste livro.

Um dos conceitos mais primordiais em programação é o de variáveis, também chamadas de objetos (em linguagens orientadas à objetos) assim como as chamadas constantes (estrutura de dados existente em algumas linguagens, ausente em outras).

Uma variável, didaticamente falando, nada mais é do que um objeto alocado em memória que guarda dentro desse espaço algum dado / valor, de forma que sempre que necessário conseguimos acessar os dados desse objeto para fazer uso dos mesmos.

Imagine que você tem um móvel com algumas gavetas, onde você guarda objetos nas gavetas de acordo com sua organização. Sempre que necessário, você abre a gaveta, pega o objeto, o usa, e após terminado o uso do mesmo, o guarda novamente em sua gaveta.

É exatamente assim que uma linguagem de programação trabalha gerenciando dados na memória do computador, abstraindo dados / valores a serem guardados de forma acessível em um espaço de nossa memória reservado para algum propósito. Durante todo o tempo de execução de nosso programa, sempre que necessário, objetos alocados em memória, com diferentes tipos de dados, são utilizados e reutilizados conforme a demanda, sendo descarregados da memória quando fechamos nosso programa.

Certamente você em algum momento já viu o gerenciador de tarefas de seu sistema operacional, onde são exibidos seus programas abertos assim como a quantidade de memória alocada para os mesmos enquanto estiverem em execução.

Uma variável recebe este nome pois como veremos logo em seguida, estaremos criando um objeto lógico com um nome personalizado onde os dados / valores atribuídos para o mesmo podem ser alterados dinamicamente quantas vezes forem necessários, irrestritamente. Por exemplo, imagine uma variável de nome pessoa1 que guarda como informação o nome ‘Fernando’. A qualquer momento podemos alterar a informação atrelada a tal variável de modo que a mesma continuará se comportando de maneira totalmente funcional em nosso código.

Em contrapartida, uma constante é um tipo de dado presente em JavaScript (ausente em outras linguagens como por exemplo Python), onde como o nome sugere, estaremos criando um objeto que terá de ser imutável durante a execução do código. Por exemplo, imagine uma constante de nome data1 que recebe como valor atribuído 2021. Diversas estruturas de nosso código poderão usar deste dado associado à constante data1, porém não terão a capacidade de alterar essa data independentemente do contexto.

Como veremos em seguida, por parte de sintaxe teremos formas de declarar tais tipos de dados de modo que o interpretador da linguagem os identificará como variável ou constante, para assim determinar seu respectivo comportamento.

Declarando Variáveis

Dado todo o contexto explicado anteriormente, vamos buscar entender tudo o que está envolvido no processo de declaração de variáveis.

```
1   // nome_da_variavel = dado/valor/atributo
2
3   let numero = 20
4   // Nome da variável: numero
5   // Dado/valor atribuído: 20
6
```

Para princípio de conversa, tanto para JavaScript quanto para outras linguagens de programação, existem algumas palavras que são reservadas ao sistema, palavras estas que atuam como marcadores / identificadores para que o interpretador execute certo processo a partir de tal identificador.

No processo de declaração de variáveis, sempre que estamos a criar uma nova variável devemos inserir o prefixo “ let “, que é uma palavra reservada ao sistema para que o interpretador saiba que naquele ponto está sendo criada uma nova variável.

Quando estamos falando de palavras reservadas ao sistema estamos nos referindo a algumas palavras-chave as quais não podemos fazer uso pois estaríamos a gerar alguma exceção. Por exemplo, o identificador let não pode ser usado como nome de uma variável pois irá gerar um erro de interpretação.

Seguindo nossa linha de raciocínio, toda variável deve ser nomeada, contendo um nome personalizado e de fácil identificação, e para este nome existem algumas regras que veremos logo em seguida sobre o que é permitido ou não visando evitar comportamentos indesejados em nossos códigos.

Em JavaScript podemos declarar uma variável e não a inicializar (como se estivéssemos apenas reservando um nome para tal variável), porém é recomendado que no momento em que

declaramos uma nova variável já realizemos a atribuição de algum dado/valor para a mesma.

Para atribuirmos um dado/valor a uma variável usamos do operador de atribuição “ = “ seguido do atributo em si que poderá ser qualquer tipo de dado/valor desde que respeitemos sua sintaxe. Apenas realizando um pequeno adendo, é quase uma unanimidade em linguagens de programação, que o símbolo = é funcionalmente um operador de atribuição. Em outras situações onde realmente estaremos usando de um operador de igualdade, o símbolo será “ == “ para criar a expressão de que uma coisa “é igual a...”.

Em nosso exemplo, let numero = 20, let é o identificador de criação de variável, numero é o nome de nossa variável e 20 é seu atributo.

Toda vez que instanciarmos tal variável, estaremos o fazendo chamando a mesma por seu nome para ter acesso a seu conteúdo, ou seja, para nosso exemplo, toda vez que fizermos referência a “numero” estaremos usando de seu valor 20 para algum propósito.

Uma vez declarada uma variável, ela internamente já possui um endereço e espaço de memória alocada para seus dados.

```
1   let numero2 = 20
2
3   numero2 = 50
4
5   let nome1 = 'Veronica'
6
7   let variavel1 = [21, 'Maria', 19.90]
8
9   console.log(numero2)
10  // Exibe em tela o conteúdo da variável numero2
11
12  console.log(nome1)
13  // Exibe em tela o conteúdo da variável nome1
14  console.log(variavel1)
15  // Exibe em tela o conteúdo da variável variavel1
16
```

Para fins de exemplo, em nosso código, logo na primeira linha declaramos uma variável de nome numero2, que recebe como atributo o valor 20.

Em seguida, na linha 3 de nosso código, instanciamos a variável criada anteriormente numero2, agora atribuindo para a mesma o valor 50.

Como dito anteriormente, uma variável tem a propriedade de ter seu conteúdo alterado conforme a necessidade, logo, no momento da criação da variável numero2 a mesma possuía o valor 20, em seguida, atualizamos o valor da mesma para 50, e isto pode ser aplicado a toda e qualquer variável.

Pela chamada leitura léxica do interpretador da linguagem, um código é lido sequencialmente, linha por linha, da esquerda para direita em cada linha. Dessa forma, a última vez que o interpretador leu o valor da variável numero2 este valor era 50.

Prosseguindo com nosso exemplo, na linha 5 de nosso código é criada uma nova variável de nome nome1, que recebe como seu atributo o nome Verônica. Da mesma forma é criada na linha 7 uma nova variável de nome variavel1 que recebe uma lista com alguns elementos. Veremos em detalhe os tipos de dados atribuídos a estas variáveis nos capítulos subsequentes, por hora, repare que o processo de declaração de variáveis é bastante simples, uma vez que respeitemos a sintaxe.

Por fim, usando da função console.log( ), inserindo entre seus parênteses o que chamamos de parâmetro da função, os nomes das variáveis criadas anteriormente, ao executar esse código nos é retornado, como esperado, apenas os dados / valores das variáveis em questão.

Por exemplo, referente a linha 9 de nosso código, onde chamamos a função console.log( ), parametrizando a mesma com a variável numero2, estamos fazendo com que o conteúdo da variável numero2 seja exibido em tela.

Fique tranquilo caso algum destes pontos por hora pareça um tanto abstrato, veremos cada detalhe envolvido no momento certo, por hora, nosso foco é no processo de declaração de variáveis, apenas por questões didáticas estou realizando a interação de nossas variáveis com uma função do sistema.

```
1    const data = 2021
2
```

Por fim, para declarar uma constante simplesmente usamos como prefixo para o nome da variável a palavra reservada ao sistema const.

Como mencionado anteriormente, uma constante deve ser usada para situações onde teremos um objeto o qual tem seu atributo fixo, imutável após sua declaração.

```
1    const data = 2021
2
3    data = 2022
4
```

```
[Running] node "c:\Users\Fernando\Desktop\JavaScript\teste.js"
c:\Users\Fernando\Desktop\JavaScript\teste.js:3
data = 2022
     ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (c:\Users\Fernando\Desktop\JavaScript\teste.js:3:6)
    at Module._compile (internal/modules/cjs/loader.js:1072:14)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1101:10)
    at Module.load (internal/modules/cjs/loader.js:937:32)
    at Function.Module._load (internal/modules/cjs/loader.js:778:12)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.
    js:76:12)
    at internal/main/run_main_module.js:17:47

[Done] exited with code=1 in 0.136 seconds
```

Simulando um erro, uma vez que temos uma constante declarada, ao tentar atualizar o valor dessa variável ao longo do código é gerado um erro e em consequência disso a execução do código é

interrompida. Posteriormente entenderemos os mecanismos de tratamento de erros, aqui, o ponto é que toda linguagem de programação possui suas regras que, quando violadas, geram erros e interrompem o andamento do código.

Uma última particularidade interessante de ser abordada a este momento é que existem ao menos duas formas de se declarar uma variável em JavaScript. Uma delas, via let, é a forma mais moderna e simplificada, enquanto é possível também declarar uma variável usando do prefixo “ var “.

Na prática, o que basicamente diferencia uma forma de declaração de outra é que através de let criamos uma variável normalmente, porém com a característica que uma vez declarada apenas podemos fazer referência a tal variável.

```
1   let pessoa1 = 'Carlos'
2
3   let pessoa1 = 'Camila'
4
5   pessoa1 = 'Maria'
6
```

```
[Running] node "c:\Users\Fernando\Desktop\JavaScript\teste.js"
c:\Users\Fernando\Desktop\JavaScript\teste.js:3
let pessoa1 = 'Camila'
    ^

SyntaxError: Identifier 'pessoa1' has already been declared
    at wrapSafe (internal/modules/cjs/loader.js:988:16)
    at Module._compile (internal/modules/cjs/loader.js:1036:27)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1101:10)
    at Module.load (internal/modules/cjs/loader.js:937:32)
    at Function.Module._load (internal/modules/cjs/loader.js:778:12)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.
    js:76:12)
    at internal/main/run_main_module.js:17:47

[Done] exited with code=1 in 0.122 seconds
```

Ao tentarmos redeclarar uma variável via let é gerado um erro nos dizendo que a variável em questão já foi previamente declarada.

```
1   var pessoa2 = 'Gabriela'
2
3   var pessoa2 = 'Tania'
4
```

Fazendo o uso de var é perfeitamente possível redeclarar uma variável. Lembrando que pela leitura léxica por parte do interpretador, o último dado / valor atribuído a uma variável é o que vale.

Em nosso exemplo, quando instanciarmos a variável pessoa2, independentemente do propósito, o último dado / valor atribuído para a mesma, que estaremos trabalhando será ‘Tania’.

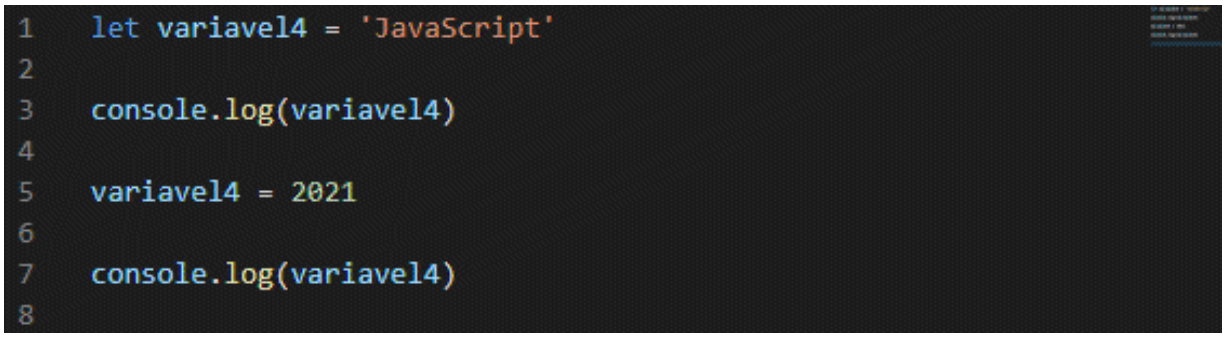
Em resumo, em JavaScript temos variáveis e constantes, cada tipo de dado com suas devidas particularidades que devemos respeitar para evitar exceções em nossos códigos.

Tipos de Variáveis

Entendido o processo de declaração de variáveis podemos avançar alguns passos, entendendo as particularidades dos tipos de dados que estaremos usando atribuídos às nossas variáveis.

JavaScript é uma linguagem de tipagem dinâmica, ou seja, quando declaramos uma variável e atribuímos um certo dado/valor para a mesma, a qualquer momento podemos alterar tais dados / valores, consequentemente alterando o tipo de dado identificado para uma variável.

Essa tipagem, comumente chamada de tipagem fraca, é muito útil no que diz respeito à flexibilidade que temos para manipular dados atribuídos às nossas variáveis. Em contrapartida, como veremos em seguida, existem situações onde podem ocorrer conflitos entre tipos de dados, situações que teremos de contornar realizando certas validações.



Por exemplo, logo na primeira linha de nosso código declaramos, de acordo com a sintaxe, uma variável de nome variavel4, que recebe como atributo um elemento textual.

Exibindo em tela através da função console.log( ) o conteúdo da variável variavel4, o retorno é JavaScript.

Já na linha 5 de nosso código instanciamos novamente a variável variavel4, agora repassando como atributo para a mesma um elemento numérico. Dessa forma, estamos atualizando o conteúdo da variável variavel4, inclusive alterando o tipo de dado atribuído.

Exibindo em tela novamente via console.log( ) o conteúdo de variavel4, agora é possível ver que seu último dado/valor associado foi 2021.

Tipo String

Um dos tipos de dados mais básicos em JavaScript (e não somente em JavaScript mas em outras linguagens como Python) são as chamadas strings, elementos textuais que por conta de sua sintaxe sempre estarão na notação de “ ‘ ‘ “ aspas.

```
1   let linguagem = 'Javascript'
2
3   console.log(linguagem)
4   console.log(typeof(linguagem))
5
```

Como exemplo, declaramos uma variável de nome linguagem que recebe como atributo uma string, um texto envolto em aspas que será identificado graças a elas como um elemento textual por parte do interpretador.

Exibindo em tela via console.log( ) o conteúdo da variável linguagem nos é retornado, conforme esperado, JavaScript.

Exibindo em tela via console.log( ) aninhada com a função typeof( ), por sua vez parametrizada com a variável linguagem, nos é retornado o tipo de dado reconhecido para tal variável, nesse caso, string.

```
1   let linguagem = 'Javascript'
2               let linguagem: string
3   console.log(linguagem)
4
```

Simplesmente posicionando o cursor do mouse em cima do nome da variável é possível ver que a mesma de acordo com sua sintaxe é identificada como do tipo string.

```
let ano = 2021

console.log(typeof(ano))

ano = '2021'

console.log(typeof(ano))

```

O principal a ser entendido neste momento é que a notação, a forma como estaremos escrevendo certos dados / valores para uma variável, será o identificador de tipo de dado para o interpretador. *Posteriormente veremos formas de manualmente definir o tipo de um dado ou converter através de funções específicas um tipo de dado em outro.

Em nosso exemplo, declarada uma variável de nome ano, que recebe 2021 como seu atributo, exibindo em tela seu tipo de dado via console.log(typeof( )) é retornado “number” pois como bem sabemos 2021 é um elemento de tipo numérico.

Alterando a sintaxe do atributo da variável ano de 2021 para ‘2021’ (mesmo número, envolto em aspas), tal dado passa a ser reconhecido como um texto/string. O que pode ser confirmado exibindo em tela o conteúdo de ano repassando a mesma como parâmetro para console.log(typeof( )).

Repare que o dado/valor se manteve o mesmo, nesse caso, 2021, porém, quando o mesmo é escrito como 2021 é reconhecido como número, e quando escrito como ‘2021’ é reconhecido como texto.

Na prática, esta notação irá influenciar no que diz respeito as possibilidades de interação com esses dados. Raciocine que entre tipos numéricos é possível realizar operações matemáticas, enquanto com textos as possibilidades são outras completamente diferentes.

```
let aviso1 = 'Não apertar o botão "DESLIGAR"'

console.log(aviso1)

```

Outro ponto a considerar é que, uma vez que aspas identificam um certo dado como do tipo string para o interpretador, quando temos de usar aspas em forma literal para um texto, devemos usar aspas diferentes das usadas como identificador.

No exemplo, declarada uma variável de nome aviso1, que recebe como atributo uma frase em forma de string, repare que para a frase em si foi usado de aspas simples, enquanto para a palavra em destaque dentro da frase foi usado aspas duplas para diferenciar tal texto do identificador.

O retorno nesse caso será: Não apertar o botão “DESLIGAR”

```
let id1 = "Marca D'água"

console.log(id1)

```

O mesmo tipo de diferenciação pode e deve ser aplicado em casos onde se faz uso de apóstrofe.

Em nosso exemplo, Marca D’água é uma string atribuída a variável id1, onde a string em si foi identificada por aspas duplas enquanto a apóstrofe da palavra é funcionalmente uma aspa simples.

Tipo Number

Como já vimos indiretamente em exemplos anteriores, é muito comum atribuirmos a uma variável um dado do tipo numérico. Basicamente, todo e qualquer número em JavaScript é do tipo “number“, diferentemente de outras linguagens de programação que separam, por exemplo, tipo inteiro de número com casa decimal para aplicar regras diferentes a cada tipo.

Por parte sintática também não existe nenhuma peculiaridade, bastando escrever o número normalmente, respeitando a notação americana de marcador de casa decimal, quando for o caso, usando “ . “ ponto como separador.

```
1   let mes = 11
2
3   console.log(mes)
4   console.log(typeof(mes))
5
6   let valor = 14.90
7
8   console.log(valor)
9   console.log(typeof(valor))
10
```

Para nosso exemplo inicialmente é criada uma variável de nome mês, que recebe como atributo o número 11.

Da mesma forma criamos uma variável de nome valor, que por sua vez recebe como atributo o número 14.90.

Exibindo em tela via console.log(typeof( )) o tipo de dado associado as variáveis mês e valor, para ambas é retornado o tipo number.

Tipo Boolean

Um dos tipos de dados que serão dos mais relevantes em nossos códigos são os chamados booleanos. Dentro da lógica booleana temos certas expressões onde o desfecho de uma determinada ação/função será um dado verdadeiro ou falso conforme o contexto.

Posteriormente veremos estruturas lógicas, estruturas condicionais e de validação, onde nos aprofundaremos nos meios e métodos de uso de dados booleanos. Neste momento, raciocine que para JavaScript temos algumas estruturas internas identificadas pelo interpretador como verdadeiro ou falso, e tais estruturas são um tipo de dado em particular.

```
1   let gatilho1 = true
2   let gatilho2 = false
3
4   console.log(gatilho1)
5   console.log(typeof(gatilho1))
6   console.log(gatilho2)
7   console.log(typeof(gatilho2))
8
```

Visualizando tais conceitos na prática, de início criamos duas variáveis de nomes gatilho1 e gatilho2, respectivamente.

Exibindo em tela os conteúdos e os tipos de dados destas variáveis através das funções console.log( ) e console.log(typeof( )), nos é retornado, como esperado, os valores lógicos das variáveis assim como seu tipo boolean.

Note que ao repassar como atributo para uma variável os booleanos true ou false, os mesmos recebem uma coloração diferente do usual pois tais nomes são palavras reservadas ao sistema. Também não existe a notação de aspas para esses elementos textuais pois os mesmos não são strings, mas tipos de dados específicos chamados booleanos.

Tipo Array

Como você deve ter reparado, os tipos de dados vistos até o momento estavam bastante engessados em ser “um dado/valor” atribuído a uma variável, mas e quando queremos guardar múltiplos dados associados a uma única variável? A resposta é por meio de arrays.

Arrays em JavaScript são o equivalente a listas em Python, onde teremos uma sintaxe própria que identificará tal tipo de dado ao interpretador, assim como podemos inserir irrestritamente elementos dentro dessa lista, de modo que cada elemento terá seu tipo de dado específico, assim como a lista em si terá um mapeamento interno que nos gera um índice pelo qual conseguimos iterar sobre um ou mais elementos específicos de dentro desse contêiner de dados.

```
1   let minha_lista = [1987, 'Fernando', true]
2
3   console.log(minha_lista)
4
```

Indo direto ao ponto, declaramos uma variável de nome minha_lista, que por sua vez recebe como atributo uma array composta de 3 elementos.

Repare na notação, uma array é identificada pelo interpretador quando temos um conjunto de dados / valores envoltos entre colchetes.

Note também que cada um dos elementos internos a esta array são de tipos diferentes de dados, cada um com sua sintaxe particular, usando de vírgula como separador entre estes elementos.

Como feito anteriormente diversas vezes, exibindo em tela o conteúdo da variável minha_lista através do método console.log( ), nos é retornado uma lista de elementos.

```
let nomes1 = ['Ana', 'Maria', 'Paulo', 'Tânia']

console.log(nomes1)
console.log(nomes1[1])

```

Como dito anteriormente, uma array é um contêiner de dados, uma lista sequencial de múltiplos elementos indexados e atribuídos a uma variável, de forma que podemos iterar sobre um ou mais elementos desta estrutura sempre que necessário.

Como exemplo, é declarada uma variável de nome nomes1, que por sua vez recebe uma array com 4 elementos em forma de string.

Exibindo em tela via console.log( ) o conteúdo de nomes1 nos é retornado toda a lista de elementos.

Anteriormente foi comentado que para este tipo de dado existe um sistema de indexação interna para seus elementos. Este mecanismo de mapeamento e indexação, embora implícito, é perfeitamente usável por parte do desenvolvedor, bastando que o mesmo apenas especifique um valor número de índice para que seja retornado o dado/valor do elemento situado naquela posição da array.

Em nosso exemplo, repassando como parâmetro para console.log( ) nomes1[1] estamos pedindo acesso ao elemento situado na posição 1 do índice, nesse caso, Maria.

Lembrando que todo e qualquer índice em JavaScript é iniciado a contar de 0, ao referenciar de nomes1 o elemento da posição de índice 1, o retorno será Maria conforme esperado.

```
let nomes1 = ['Ana',
              'Maria',
              'Paulo',
              'Tânia']

console.log(nomes1)
console.log(nomes1[1])

```

Apenas salientando que por questões de legibilidade, os elementos de uma lista (ou de qualquer outro contêiner de dados) podem ser declarados em múltiplas linhas, desde que se respeite a sintaxe dos mesmos e da própria lista, que usa como separador / quebra de linha o símbolo de vírgula.

```
let nomes1 = ['Ana',
              'Maria',
              'Paulo',
              'Tânia']

nomes1.push('Fernando')

console.log(nomes1)

```

Mantendo a mesma estrutura de código do exemplo anterior, uma vez que temos nossa array nomes1 já declarada e estabelecida, para adicionar novos elementos a esta array fazemos o uso da função push( ), parametrizada com o dado/valor a ser inserido na array.

Repare na sintaxe, aqui estamos instanciando a variável nomes1 declarada anteriormente e aplicando sobre a mesma uma função, nesse caso push( ) para assim inserir um novo elemento na array.

Usando deste método, o novo elemento, independentemente de seu tipo, será inserido por padrão na última posição da array. Em outras palavras, uma vez que temos uma estrutura de múltiplos elementos sequenciais, ao inserir um novo elemento o mesmo será posicionado ao “final da fila”.

Executando esse bloco de código será possível verificar que de fato ‘Fernando’ foi inserido como 5º elemento da array atribuída a nomes1.

```
let nomes1 = ['Ana',
              'Maria',
              'Paulo',
              'Tânia']

nomes1.pop('Fernando')

console.log(nomes1)

```

De modo parecido, podemos através da função pop( ) remover um elemento de uma array. Por padrão, ao aplicar tal método sobre uma variável, o último elemento da array atribuída para a mesma será removido.

No nosso exemplo, ‘Fernando’ que havia sido inserido anteriormente foi o elemento removido da array.

```
let nomes1 = ['Ana',
              'Maria',
              'Paulo',
              'Tânia']

console.log(nomes1)
console.log(typeof(nomes1))

```

Uma última particularidade a ser entendida neste momento é que, ao se verificar o tipo de dado de uma array em JavaScript obtemos como retorno “ object “, que é um tipo de dado primitivo padrão para contêineres de dados / estruturas onde temos múltiplos dados atrelados a uma única variável.

Tipo Object

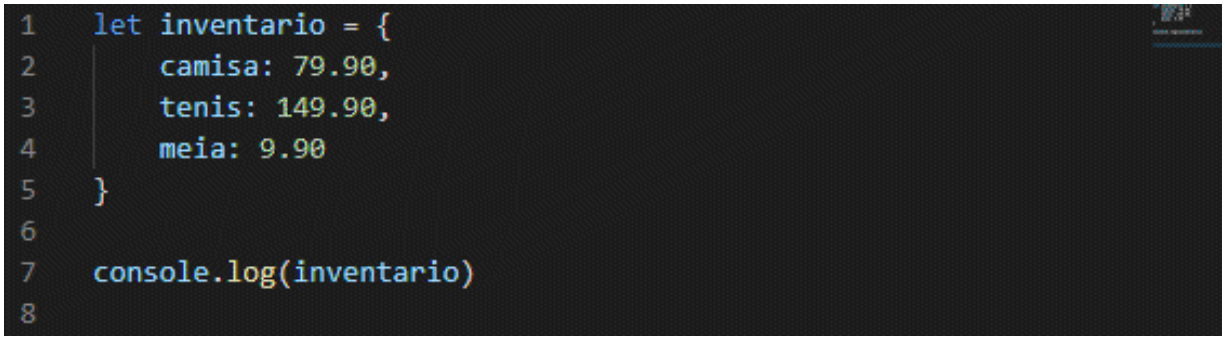
Avançando um pouco mais em nossos estudos, vamos buscar entender outro tipo de dado bastante comum e robusto em JavaScript chamado object.

Equivalente a um dicionário em outras linguagens de programação, um object em JavaScript é uma estrutura de dados onde dentro da mesma podemos guardar itens organizados em chave:valor.

A notação sintática de um object se dá pelo uso de “ { } “ chaves, que demarcam e identificam para o interpretador que ali existe um contêiner de dados organizados em chaves e valores.

Dentro dessa lógica, estaremos criando dentro de uma estrutura atribuída a uma única variável, múltiplos itens que recebem um nome o qual chamamos de chave, que quando instanciado retornará algum atributo o qual chamamos por convenção de valor.

Assim como em uma array, um object pode receber todo e qualquer tipo de dado em sua composição desde que respeitemos sua sintaxe.



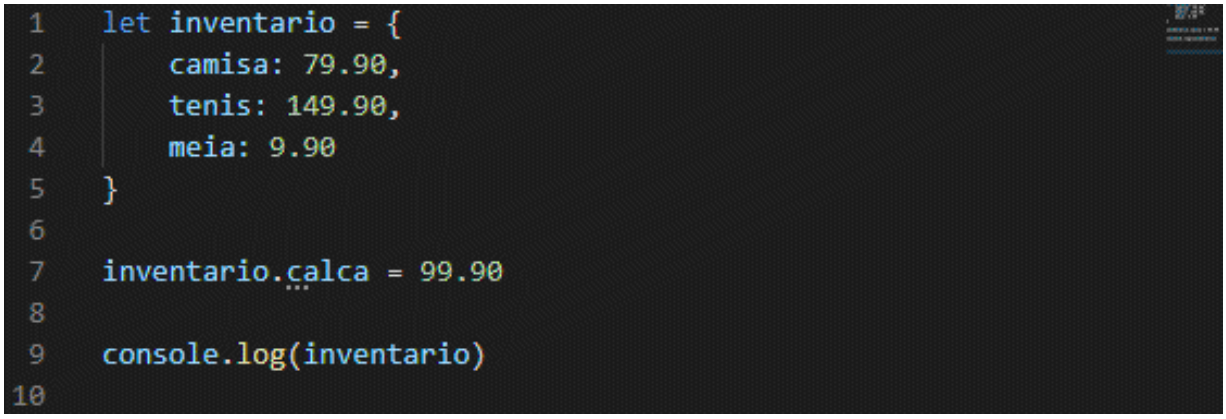
```
1  let inventario = {
2      camisa: 79.90,
3      tenis: 149.90,
4      meia: 9.90
5  }
6
7  console.log(inventario)
8
```

Partindo para a prática, na primeira linha de nosso código é declarada uma variável de nome inventario, que por sua vez recebe como atributo um object composto de alguns elementos organizados em chaves:valores.

Repare que por parte de sintaxe o object em si é delimitado por chaves, assim como cada elemento interno é separado por vírgula. Por uma questão de organização e legibilidade do código, uma

prática comum é inserir cada item de nosso object em uma linha própria.

Através da função console.log( ), por sua vez parametrizada com a variável inventario, é exibido em tela o conteúdo da mesma.



```
let inventario = {
    camisa: 79.90,
    tenis: 149.90,
    meia: 9.90
}

inventario.calca = 99.90

console.log(inventario)

```

Para inserir um novo item em nosso object, simplesmente instanciamos a variável a qual o object está atribuído, inserindo um novo objeto com seu respectivo dado/valor.

Visualmente é como se estivéssemos inserindo dentro de um contêiner de dados uma nova variável com seu respectivo atributo, porém, devido o contexto desse tipo de dado, aqui estamos a inserir uma nova chave com seu respectivo valor.

Tipo Function

Por fim, uma das estruturas fundamentais em JavaScript, e que pode causar uma certa confusão em quem está aprendendo a programar do zero, é o fato de que internamente tudo em JavaScript é uma função.

Por mais abstrato que isso conceitualmente possa parecer neste momento, raciocine que cada tipo de dado em JavaScript é condicionado de modo a executar certas ações dentro do propósito do código.

No capítulo seguinte estaremos a criar nossas primeiras funções personalizadas, entendendo a lógica de execução de uma função, mas por hora tenha em mente que cada tipo de objeto em um código escrito para JavaScript está abstraindo algo em uma função.

```
1   console.log(typeof(Object))
2
```

Ao longo dos exemplos apresentados até este capítulo, por diversas vezes usamos as funções console.log( ) e typeof( ) para respectivamente exibir em tela o conteúdo de uma variável assim como seu tipo de dado.

A função console.log( ) é o que chamamos basicamente de uma função de saída, pois está a partir do código gerando um retorno ao usuário, enquanto a função typeof( ) é uma função interna do sistema que simplesmente recebe um objeto qualquer, inspeciona o mesmo retornando seu tipo de dado, o que será útil posteriormente para certas situações.

Retomando a linha de raciocínio deste tópico, em nosso exemplo, chamando diretamente a função console.log( ), parametrizada com a função typeof( ), por sua vez parametrizada com Object (palavra reservada ao sistema para todo e qualquer objeto genérico em JavaScript), obtemos como retorno “ function “, pois mesmo a estrutura pré-moldada mais básica de um programa em JavaScript

internamente possui as propriedades de um objeto pronto para executar alguma ação.

Quando declaramos uma variável, por exemplo, existem uma série de mecanismos internos sendo executados preparando um objeto com um rótulo, um espaço alocado em memória, um tipo de dado a ser identificado e uma possível ação a ser realizada.

Da mesma maneira, todo e qualquer tipo de estrutura de dados em JavaScript em alguma camada de sua estrutura possui essas mesmas características. Sendo assim, a nomenclatura function costuma confundir um pouco quando estamos a aprender os conceitos básicos dessa linguagem, embora na prática este tipo de dado é algo implícito ao desenvolvedor.

Assim como em Python tudo é objeto (até mesmo funções), em JavaScript tudo é função (até mesmo objetos), embora ambas as linguagens em sintaxe e funcionamento sejam muito parecidas.

```
1   class Pessoa {}
2
3   console.log(typeof(Pessoa))
4
```

Apenas finalizando este tópico, usando de um outro exemplo, criada uma classe de nome Pessoa, sem nenhuma estrutura de código definida, ao exibir em tela o tipo de dado deste objeto nos é retornado function. À medida que estruturas forem sendo criadas para esta classe, com seus respectivos objetos / variáveis e tipos de dados definidos, certos comportamentos serão adotados para tal objeto.

Raciocine que uma vez criado um objeto primitivo em JavaScript, o mesmo possui um leque aberto com todas as possíveis ações a serem realizadas pelo mesmo, conforme definimos o tipo de objeto e seu respectivo tipo de dado, o mesmo passa a se “especializar” em uma determinada ação.

Como mencionado anteriormente, tais conceitos são um tanto quanto abstratos neste momento, sendo assim, não se preocupe

pois no momento certo estaremos retomando estes conceitos e aumentando nossa bagagem de conhecimento sobre os mesmos.

Por hora, entenda que um dos tipos de dados fundamentais em JavaScript é o chamado function.

FUNÇÕES BÁSICAS

Uma vez entendidos os conceitos básicos sobre sintaxe e tipos de dados, podemos subir alguns degraus partindo para o uso de funções, para que (desculpe a redundância) possamos implementar funcionalidades em nossos códigos.

Quando estamos programando, independentemente de qual seja o propósito, em nosso código estaremos escrevendo por meio de sentenças instruções a serem interpretadas pelo núcleo da linguagem para que assim se criem objetos lógicos e funções que interagem com os mesmos executando tarefas específicas.

Os chamados algoritmos nada mais são do que scripts onde abstraímos determinados problemas computacionais em instruções a serem cumpridas em etapas por nosso programa.

Dentro dessa linha de raciocínio, toda linguagem de programação já vem com certas estruturas pré-moldadas para que possamos interagir com o sistema operacional (e consequentemente o hardware que temos à disposição) por meio de instruções escritas em linguagem de alto nível. Lembrando que quando falamos em “nível” de uma linguagem de programação nos referimos a camada a qual estamos passando instruções para o computador, JavaScript por exemplo é uma linguagem de alto nível, onde a linguagem usada está mais próximo de nossa linguagem natural do que da linguagem de máquina.

Todo código JavaScript executado passa por camadas de interpretação até chegar em linguagem de base como Assembly que é finalmente “traduzida” para linguagem de máquina mesmo, instruções a nível dos registradores lógicos de um processador, em sequências binárias a serem processadas por circuitos lógicos (e físicos em sua etapa final).

O ponto é que programar, usando das ferramentas certas, na camada certa, é instruir o computador a realizar certas ações.

A nível de código, estaremos escrevendo funções para realizar determinadas tarefas usando de nossas estruturas lógicas dispostas em variáveis, e dentro dessa premissa, a estrutura de uma função é um trecho de código, também com suas particularidades no que tange a sintaxe, com cada uma das etapas a serem realizadas no processo, de modo que uma função pode ou não gerar um retorno, assim como uma função pode ser escrita para uma única funcionalidade pontual ou de forma que possa ser reutilizada quantas vezes sua ação for necessária.

Como nas aulas de algoritmo nos é ensinado, o script em si pelo qual uma função é escrita é como uma receita de bolo, onde se recebem certos ingredientes, com suas devidas proporções e ordem de inclusão, em seguida processando os mesmos de acordo com uma série de critérios para no resultado final se ter de fato um bolo. Jogar todos os ingredientes em um recipiente não faz um bolo, é necessário seguir uma sequência lógica dos fatos, passo a passo, sequencialmente, para que no final do processo se tenha um bolo.

Cada etapa envolvida na confecção de um bolo é uma ação dentro de um propósito maior que é a ação de se fazer um bolo, e na prática é esta mesma linha de raciocínio que estaremos fazendo uso quando estivermos a escrever funções para nossos programas.

Funções Embutidas

Como função “embutida” entende-se funções prontas já inclusas no núcleo da linguagem. Como já usamos diversas vezes ao longo de nossos exemplos, a função console.log( ) é uma função embutida com propósito de exibir em tela / terminal o conteúdo de uma variável / objeto.

Caso tal função não existisse na base da linguagem, teríamos de escrever do zero uma função que ganhasse acesso ao console, para nele exibir o conteúdo de uma variável dentro de uma série de etapas a serem realizadas internamente (inclusive englobando outras funções do sistema), de modo que a simples ação de exibir em tela o conteúdo de uma variável seria algo maçante.

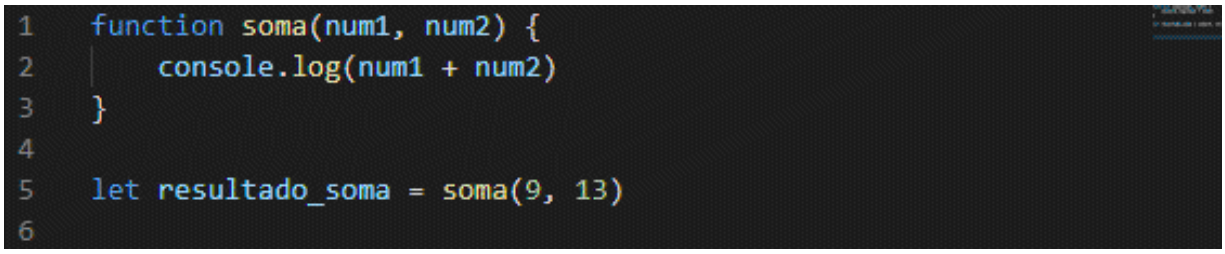
Toda linguagem de programação traz uma série de funções pré-moldadas onde basta o desenvolvedor ver conforme a documentação da linguagem como se dá o uso da função a qual o mesmo necessita usar.

Funções Personalizadas

Diferentemente do conceito anterior, toda linguagem de programação permite que o desenvolvedor crie suas próprias funções personalizadas, haja visto que seria praticamente impossível para a equipe de desenvolvimento do núcleo da linguagem programar todas as possíveis funções com todas as possíveis possibilidades de uso para todos os possíveis contextos.

Logo, por parte da linguagem de programação, é oferecida ao desenvolvedor ferramentas para que de modo intuitivo o mesmo crie suas próprias funções.

Por parte de sintaxe, toda função deve respeitar algumas particularidades para que seja reconhecida como tal assim como executadas corretamente suas ações programadas.



Diretamente na prática, inicialmente é essencial entender que uma função em JavaScript, assim como todas as demais estruturas de dados vistas até então, possui uma sintaxe própria a ser respeitada.

Para criarmos uma função personalizada usamos do identificador function, seguido do nome personalizado da função, seguido de um campo para inserção de parâmetros, demarcado por parênteses, seguido de um campo para escrita de todas as sentenças que realizam as devidas etapas da função, delimitado por chaves.

Observando o exemplo, de acordo com a sintaxe criamos uma função de nome soma( ), que receberá dois parâmetros a partir de variáveis ou de algo digitado pelo usuário, repassando tais dados para num1 e num2. A partir deste ponto, é exibido em tela através de console.log( ) a soma dos dois valores repassados para num1 e num2.

Na sequência, fora da função, de volta ao corpo / escopo global de nosso código criamos uma variável de nome resultado_soma, que por sua vez “chama” a função soma, repassando como parâmetros para a mesma os valores 9 e 13.

Ao executar este bloco de código é gerado um retorno via terminal com o valor 22, valor retornado do processamento de nossa função soma( ).

Importante salientar que, da maneira com que nossa função soma( ) foi escrita, sua “função” era apenas receber dois valores e exibir em tela o resultado da soma dos mesmos.

Uma outra possibilidade que temos é escrever uma função que retorna um dado/valor, de modo que possamos reutilizar esse valor para outros propósitos.

```
1  function soma(num1, num2) {
2      return num1 + num2
3  }
4
5  let res_soma = soma(5, 12)
6
7  console.log(res_soma)
8
```

Partindo para outro exemplo, definida a função soma( ), alteramos no corpo da mesma a ação a ser realizada para return num1 + num2. Dessa forma, via return (palavra reservada ao sistema para retorno de dado/valor de uma função) o valor resultante da soma de num1 e num2 será guardado em memória, mesmo que temporariamente, para que tal valor possa ser utilizado por alguma variável ou até mesmo outra função.

Em seguida no corpo geral do código é declarada uma variável de nome res_soma que instancia e inicializa a função soma( ), parametrizando a mesma com 5 e 12, valores para num1 e num2, respectivamente.

Na sequência, repare que usando de console.log( ) exibimos o conteúdo da variável res_soma, nesse caso, 17. O ponto é que quando a variável res_soma chama a função soma( ), repassando os devidos valores para a mesma, ela terá como atributo final o valor do retorno da função. Supondo que outras estruturas posteriores no código iterassem sobre a variável res_soma, estariam acessando o valor 17 salvo na mesma.

Em outras palavras, no momento em que o interpretador lê a linha 5 de nosso código de fato todo o mecanismo envolvido na função soma( ) é executado, gerando um retorno que ficará atribuído a variável res_soma. A partir desta linha, sempre que outro objeto iterar sobre res_soma, terá acesso a seu último valor atribuído, nesse caso, 17.

OPERADORES

Como visto em tópicos anteriores, em programação trabalhamos com variáveis que nada mais são do que espaços alocados de memória para que possamos armazenar dentro destes objetos lógicos algum dado/valor o qual faremos uso ao longo de nosso programa.

Estes dados por sua vez, independentemente do seu tipo, recebem uma nomenclatura personalizada e particular que nos permite os identificar facilmente em meio ao código, da mesma forma, grande parte das IDEs costumam atribuir diferentes cores para cada tipo de elemento do código para também facilitar sua identificação.

Em meio a isto temos o que chamamos de operadores, símbolos que são reservados ao sistema e que contém alguma propriedade / função dentro do código.

O primeiro operador que devemos conhecer é o de atribuição, nomenclatura usada para quando queremos associar um determinado dado/valor a uma variável / constante.

Operadores básicos

Operador = para atribuições

```
1   let numero2 = 20
2
3   numero2 = 50
4
5   console.log(numero2)
6
```

Reutilizando um bloco de código de um exemplo anterior, repare que na primeira linha deste código estamos a declarar uma variável de nome numero2, que recebe o valor 20 como seu atributo.

Note que pela sintaxe, usamos de let para declarar a variável, em seguida nomeamos a mesma com um nome personalizado, seguido do símbolo “ = “, que é um operador de atribuição e do dado/valor a ser associado / atribuído a nossa variável numero2.

O símbolo de igual = em JavaScript é um operador de atribuição, ou seja, este símbolo sempre que localizado em uma estrutura de código significa que um determinado dado/valor está sendo atrelado a uma variável. Como veremos em capítulos posteriores, “ = “ é um operador de atribuição, enquanto em casos onde precisamos usar deste símbolo como um operador de igualdade para alguma sentença ou expressão matemática, faremos uso do mesmo no formato “ == “.

Por fim, repare que no bloco de código de nosso exemplo temos na linha 5 a instância da função console.log( ), que não recebe atributos, mas parâmetros, logo, para a mesma não há necessidade de usar o operador de atribuição, haja visto que toda e qualquer função em JavaScript possui um campo onde são inseridos os parâmetros, delimitado por parênteses.

Operador + para Soma ou Concatenação

```
1   console.log(128 + 99)
2
```

Dos símbolos reservados ao sistema, o de soma é identificado em JavaScript como um operador de soma ou de concatenação. De modo geral, quando estivermos usando do operador soma com números, automaticamente estaremos realizando a soma dos mesmos, já para outros tipos de dados, como textos, o operador + irá simplesmente concatenar / juntar tais textos.

No exemplo acima, repassando diretamente como parâmetro para a função console.log( ) a expressão 128 + 99 o retorno será o resultado da soma dos mesmos.

```
1   let num1 = 2021
2   let num2 = 55
3
4   const num3 = num1 + num2
5
6   console.log(num3)
7
```

Outra possibilidade é usar do operador de soma em variáveis, pois como entendido previamente, ao usarmos de uma ou mais variáveis na verdade sempre estaremos trabalhando com seu dado/valor associado.

Como exemplo, inicialmente são declaradas duas variáveis de nome num1 e num2, que recebem como atributo os valores 2021 e 55, respectivamente.

Em seguida é criada uma constante de nome num3 que recebe como atributo o resultado da soma dos valores de num1 e num2. Repare que a expressão matemática envolvida aqui na verdade está usando dos valores atribuídos as variáveis num1 e num2 e do operador + para somar tais valores.

Por fim, exibindo em tela o conteúdo da constante num3 obtemos, como esperado, o valor de 2076.

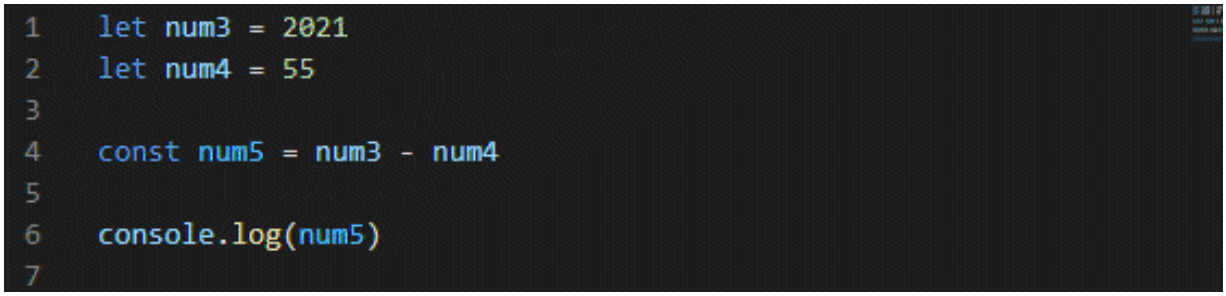
```
let nome1 = 'Fernando'
let nome2 = 'Feltrin'

let nome_completo = nome1 + nome2

console.log(nome_completo)

```

Já em situações onde não estamos trabalhando com dados / valores em formato numérico, o operador soma irá apenas concatenar os textos.

Nesse caso, o resultado da concatenação dos dados de nome1 com nome2, atribuído a variável nome_completo será Fernando Feltrin.

Operador – para subtração

De forma muito parecida com o operador soma, o de subtração pode ser usado diretamente para obter a diferença entre dois números, assim como pode ser usado em uma expressão a partir de variáveis.



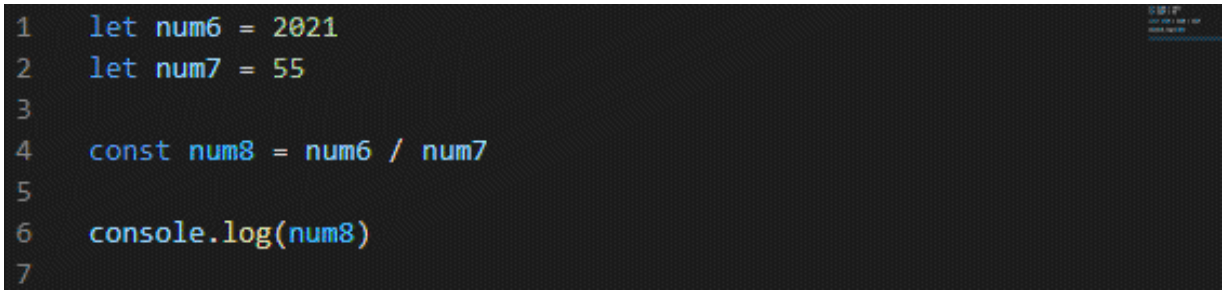
```
let num3 = 2021
let num4 = 55

const num5 = num3 - num4

console.log(num5)

```

Para nosso exemplo agora declaramos duas variáveis de nomes num3 e num4, com seus respectivos valores, também declaramos uma constante de nome num5 que recebe como atributo a subtração dos valores entre num3 e num4.

Exibindo em tela o conteúdo da constante num5 nos é retornado o valor 1966.

Operador / para divisão simples

Operações de divisão inteira podem ser feitas por meio do operador /, também diretamente sobre números ou a partir de valores atribuídos à variáveis, desde que se forneçam valores para os chamados dividendo e divisor.



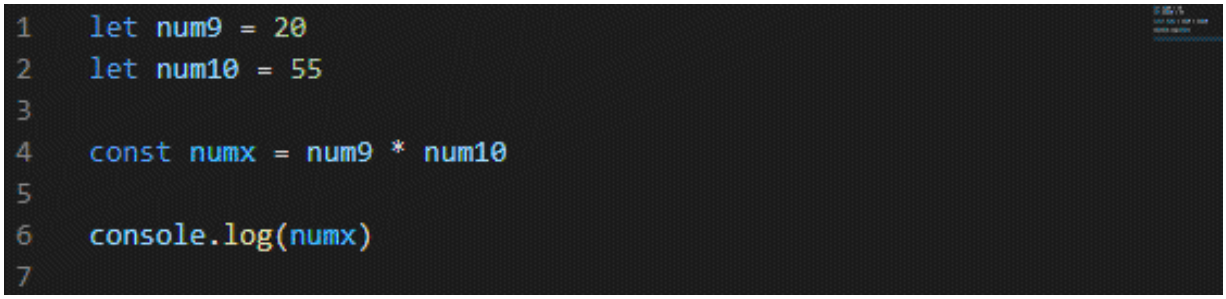
```
let num6 = 2021
let num7 = 55

const num8 = num6 / num7

console.log(num8)

```

Nos mesmos moldes dos exemplos anteriores, são declaradas as variáveis num6 e num7 com seus respectivos valores, assim como a constante num8 que recebe como atributo o módulo / resultado da divisão entre os valores de num6 e num7.

Nesse caso, o retorno será 36.74545454545454, referente à divisão entre 2021 por 55.

Operador * para multiplicação

Por último, mas não menos importante que os demais operadores das expressões matemáticas básicas temos o operador de multiplicação, representado pelo símbolo *.



```
1   let num9 = 20
2   let num10 = 55
3
4   const numx = num9 * num10
5
6   console.log(numx)
7
```

Reutilizando a mesma estrutura dos exemplos anteriores, realizando apenas as devidas alterações de nomes de variáveis e do operador usado para a expressão da linha 4 do código, obtemos como atributo para a constante numx o resultado da multiplicação dos valores das variáveis num9 e num10.

Exibindo em tela via console.log( ) o conteúdo da constante numx, nos é retornado o valor 1100.

Sendo assim, realizamos uma abordagem básica dos 4 operadores matemáticos mais comumente utilizados em aplicações. Existem outros operadores especiais para situações mais específicas, os quais veremos posteriormente.

Por hora, tenha em mente que da simbologia própria da linguagem JavaScript temos símbolos que usamos como operadores para realizar certas funções matemáticas usando dos recursos do núcleo da linguagem.

Operadores especiais

Atribuição aditiva

```
1   let numero2 = 20
2
3   numero2 = numero2 + 5
4
5   console.log(numero2)
6
```

Complementar aos operadores básicos vistos anteriormente, outra possibilidade interessante é fazer uso de operadores especiais para contextos onde se exige o uso dos mesmos para tornar nossos códigos reduzidos.

Diretamente no exemplo, na primeira linha é declarada uma variável de nome numero2, que recebe como seu atributo o valor 20. Como mencionado anteriormente, a partir do momento que o interpretador lê a linha de código referente a esta variável, implicitamente é gerado um espaço alocado em memória para que no mesmo se guarde o valor 20, a ser utilizado sempre que necessário através do rótulo de sua variável.

Na sequência, apenas para fins de exemplo, instanciamos novamente a variável numero2 para atualizar seu valor, dessa vez, através da soma de seu próprio valor acrescido de 5 através do operador.

```
1   let numero2 = 20
2
3   numero2 += 5
4
5   console.log(numero2)
6
```

A mesma expressão escrita no bloco de código anterior pode ser reduzida usando do operador +=, pois a lógica interna de leitura

dessa variável e seus dados / valores permanece com o mesmo comportamento.

Atribuição subtrativa

```
1   let numero2 = 20
2
3   numero2 -= 5
4
5   console.log(numero2)
6
```

Nos mesmos moldes do exemplo anterior, usando do operador -= estamos a realizar uma subtração.

Como exemplo, na linha 3 do código, numero2 recebe como atributo o resultado da soma de seu próprio valor (20) por 5.

Exibindo em tela o conteúdo da variável numero2 através da função console.log( ), o resultado será 15.

Atribuição multiplicativa

```
1   let numero2 = 20
2
3   numero2 *= 5
4
5   console.log(numero2)
6
```

Da mesma forma, usando do operador *= podemos instanciar a própria variável, multiplicando seu dado/valor por um outro número.

Nesse caso, numero2 tem seu valor atualizado com o resultado da multiplicação de seu valor inicial (20) por 5, resultando no número 100.

Atribuição divisiva

```
let numero2 = 20

numero2 /= 5

console.log(numero2)

```

Por fim, utilizando do operador /= podemos realizar a divisão inteira entre o último valor atribuído a uma determinada variável e um valor de dividendo.

Exibindo em tela via console.log( ) o conteúdo da variável numero2 obtemos como retorno o valor 4.

Módulo (resto de uma divisão)

```
let numero2 = 20

numero2 %= 5

console.log(numero2)

```

Outro operador especial, útil para contextos onde precisamos do valor do resto de uma divisão, é o operador de módulo representado por %=.

Anteriormente usando do operador /= realizávamos a divisão entre dois valores, um atributo de uma variável e um número referência.

Agora, usando de %= podemos extrair a informação do resto de uma divisão (quando houver).

Em nosso exemplo, a variável numero2 recebe como atributo o módulo da divisão entre seu último valor (20) e 5, resultando em 0

pois esta é uma divisão sem resto.

Novamente exibindo em tela o conteúdo de numero2 através da função console.log( ), é retornado o valor 0.

Exponenciação

```
1   let numero2 = 20
2
3   numero2 **= 5
4
5   console.log(numero2)
6
```

Diferentemente do mecanismo de multiplicação, como exponenciação entendemos um determinado valor multiplicado por algumas vezes o próprio valor de acordo com algum critério.

Em outras palavras, se tratando de exponenciação, estamos a elevar um determinado número a alguma potência.

No exemplo, usando do operador **= estamos multiplicando o valor da variável numero2 por ela mesma 5 vezes de acordo com o critério.

Operações com dois ou mais operandos

```
1   let numero3 = 20
2   let numero4 = 13
3   let numero5 = 8
4
5   const num = numero3 + numero4 * numero5
6
7   console.log(num)
8
```

Complementar aos tópicos anteriores, importante salientar que realizar operações matemáticas por meio de operadores, seja diretamente sobre números ou a partir de valores atribuídos a variáveis, não implica em nenhuma limitação.

É perfeitamente possível criar expressões usando de dois ou mais operandos, desde que respeitando sua sintaxe.

Lembrando que aqui todas as regras matemáticas gerais se aplicam, logo, em primeiro lugar serão realizadas as operações de multiplicação e divisão, para por último serem realizadas as operações de soma e subtração.

Em nosso exemplo, data a sentença numero3 + numero4 * numero5, primeiramente será realizada a multiplicação dos valores entre numero4 e numero5, por fim somando este valor com o valor de numero3.

Mais uma vez exibindo em tela o conteúdo de uma variável, nesse caso num, através da função console.log( ), o resultado é 124.

```
1   let numero3 = 20
2   let numero4 = 13
3   let numero5 = 8
4
5   const num = (numero3 + numero4) * numero5
6
7   console.log(num)
8
```

Em contextos onde propositalmente queremos que uma determinada soma ou subtração seja executada antes de uma multiplicação ou divisão, devemos encapsular os operandos entre parênteses para que assim sua expressão matemática seja executada de forma independente.

Operadores lógicos

Se tratando dos códigos que iremos desenvolver, independentemente da aplicação, muitas das vezes iremos nos deparar com situações onde são escritas expressões lógicas.

Por expressões lógicas, raciocine que a nível de código podemos ter interações entre variáveis (e demais estruturas) onde estaremos comparando uma com outra, verificando dados de uma em relação a outra, validando estruturas de código de acordo com certos critérios, entre outras aplicações, e tudo isso pode ser feito de modo reduzido fazendo uso dos operadores corretos.

Quando estamos estudando lógica de programação, um dos tópicos abordados é o que se refere ao uso de tabela verdade para equivaler ou equiparar dados.

Todos esses mecanismos lógicos são perfeitamente abstraídos como ferramentas possíveis de implementação em um código, por meio de operadores dedicados a estes fins, nesse caso && referente a and e || referente a or.

```
1   console.log(7 != 3)
2
```

Mesmo que:

```
1   let num1 = 7
2   let num2 = 3
3
4   console.log(num1 != num2)
5
```

Por exemplo, se executarmos diretamente em um console a expressão 7 != 3 receberemos como retorno o valor True, pois de acordo com a expressão, 7 diferente de 3 é uma proposição verdadeira.

Operador and

```
1    console.log(7 != 3 && 2 > 3)
2
```

Usando de duas sentenças e equiparando as mesmas por meio de um operador lógico and, por exemplo, o interpretador da linguagem irá considerar o retorno de cada sentença/expressão como um valor lógico True ou False, comparando os mesmos ao final do processo como em uma tabela verdade.

Apenas complementando nossa linha de raciocínio, executando em um console a expressão 7 != 3 && 2 > 3, o retorno será False, pois 7 é diferente de 3 (retornando True) mas 2 não é maior que 3 (retornando False), o que de acordo com a tabela verdade comum, True e False é igual a False. *Uma das proposições sendo False já invalida toda a expressão.

Operador or

```
1    console.log(7 != 3 || 2 > 3)
2
```

Usando do operador || referente a or, basta uma das proposições serem verdadeiras para que a expressão seja retornada como True. Nesse caso, 7 é diferente de 3 (True), 2 não é maior que 3, mas como a primeira proposição foi verdadeira, todo o conjunto é validado como True.

Apenas como referência, tabela verdade AND

V E V = V
V E F = F
F E V = F
F E F = F

Lembrando que neste tipo de tabela verdade, independentemente do número de proposições, basta uma ser False para invalidar todas as demais.

V e V e V e V e V e V = V
V e F e V e V e V e V = F

Tabela verdade OR

V e V = V
V e F = V
F e V = V
F e F = F

Como dito anteriormente, no uso deste tipo de tabela, basta uma das proposições ser True para validar todo o restante como verdadeiro.

V e V e V e V e V e V e V = V
F e F e F e F e F e F e F = F
F e F e F e F e F E F e V e F = V
F e F e F e F e F e F e F e F = F

Tabela verdade XOR (OR exclusivo)

V e V = F
V e F = V
F e V = V
F e F = F

Neste tipo de tabela verdade, dois de mesmo tipo de proposição são False, nenhum também é False, apenas validando como True se um ou se outro elemento da expressão for True.

Operadores relacionais

Como o próprio nome sugere, para certos contextos podemos usar de operadores que verificam e interpretam a relação entre dois dados (atribuídos ou não a variáveis).

Como já visto anteriormente em alguns exemplos, se tratando de operadores relacionais, temos os operadores > (maior que), < (menor que), >= (maior ou igual a), <= (menor ou igual a), == (igual a), != (diferente de), que nos serão muito úteis em expressões lógicas

```
1   let num1 = 8
2
3   console.log(num1 > 5)
4
```

Uma forma muito usada de se abstrair uma expressão lógica é ler a mesma em forma de pergunta. Em nosso exemplo, o valor da variável num1 (8) é maior que 5? O retorno será True.

```
1   let num1 = 8
2
3   console.log(num1 >= 8)
4
```

Na mesma lógica, o valor de num1 é maior ou igual a 8? O retorno será True.

```
1   let num1 = 8
2
3   console.log(num1 < 5)
4
```

O valor de num1 é menor que 5? O retorno será False.

```
1   let num1 = 8
2
3   console.log(num1 <= 8)
4
```

O valor de num1 é menor ou igual a 8? O retorno será True, pois 8 não é menor, mas é igual a 8, de acordo com a tabela verdade

comum, False e True = True.

```
let num1 = 8

console.log(num1 == 5)

```

O valor de num1 é igual a 5? O retorno será False.

```
let num1 = 8

console.log(num1 != 5)

```

O valor de num1 é diferente de 5? O retorno será True.

Operadores de incremento e de decremento

Ainda se tratando de operadores, símbolos reservados ao sistema com algum propósito específico, temos os chamados operadores de incremento (que atualizam o valor de uma variável somando uma unidade ao valor original) e de decremento (que atualizam o valor de uma variável subtraindo uma unidade de seu valor original), representados por ++ e --, respectivamente.

```
1   let num1 = 8
2
3   num1++
4
5   console.log(num1)
6
```

Declarada uma variável de nome num1, que recebe como atributo o valor 8, ao instanciar novamente tal variável ao longo do código, aplicando diretamente sobre a mesma o operador ++, será realizado um incremento.

Exibindo em tela via console.log( ) o último valor salvo para a variável num1, o retorno será 9.

```
1   let num1 = 8
2
3   num1++
4
5   console.log(num1)
6
7   --num1
8
9   console.log(num1)
10
```

Trabalhando no mesmo exemplo, note que na linha 3 de nosso código num1 recebe um incremento, passando a ter seu valor como 9, já na linha 7 a mesma recebe um decremento, voltando a ter como valor 8.

Exibindo em tela o conteúdo da variável num1, através da função console.log( ), será retornado 8, o último valor atribuído para a variável num1.

Operador estritamente igual

Para certos casos, pode ser necessário que uma expressão lógica seja validade inclusive considerando o modo de leitura léxica de uma expressão.

Como visto logo nos capítulos iniciais, como leitura léxica podemos resumir a forma como o interpretador da linguagem lê e processa as instruções dos códigos, e isto se dá sequencialmente, linha após linha, da esquerda para direita, elemento por elemento da linha.

Sendo assim, apenas visualizando esta possibilidade, podemos obter diferentes retornos de uma expressão lógica conforme a forma com que inserimos operadores na mesma.

```
1   let num1 = 8
2   let num2 = 9
3
4   console.log(++num1 === num2)
5
```

Para fins de exemplo, são declaradas duas variáveis, num1 e num2 com seus respectivos valores, em seguida através da função console.log( ) vamos exibir em tela o retorno de uma expressão lógica onde, de acordo com a sintaxe, o valor de num1 é imediatamente incrementado, já em seu momento de leitura assumindo o valor 9 em função da forma com que o operador de incremento foi inserido.

Realizando a comparação, verificando se no momento da leitura da expressão os valores de num1 e de num2 eram iguais (via operador ===), o retorno será True.

```
1   let num1 = 8
2   let num2 = 9
3
4   console.log(num1++ === num2)
5
```

Alterando a posição do operador de incremento e realizando a comparação estrita via ===, agora será retornado False, pois pela leitura léxica, no momento da leitura da variável num1 na expressão, a mesma ainda possuía o valor 8 antes de ser incrementada para a validação.

O ponto é que por parte de leitura léxica, em certos contextos específicos, temos que nos atentar a forma como a declaração de certos operadores pode alterar o comportamento do código.

A partir de nosso exemplo podemos visualizar que, ++num1 é lido como “o incremento do valor de num1”, enquanto num1++ é lido pelo interpretador como “valor de num1, incrementado na sequência em uma unidade”.

O operador estritamente igual === faz com que a leitura de uma determinada expressão lógica seja feita de forma literal.

Operador ternário

Em JavaScript, o chamado operador ternário, simbolizado por ?, é o tipo de operador que possui no momento de sua declaração três operandos, o que permite que a partir do mesmo se crie estruturas condicionais (que veremos em capítulos futuros) de forma reduzida. Por hora, vamos nos ater a sua simbologia e forma de escrita.

```
1   let nota = 8
2
3   let media = nota >= 7 ? 'Aprovado' : 'Reprovado'
4
5   console.log(media)
6
```

Logo na primeira linha é declarada uma variável de nome nota que recebe um valor como seu atributo.

Na sequência é declarada uma nova variável, esta de nome media, que pro sua vez recebe como atributo o resultado de uma expressão ternária, escrita por nota >=7 ? ‘Aprovado’ : ‘Reprovado’.

Vamos buscar entender a expressão ternária, que por sua vez obrigatoriamente recebe três elementos / operandos, sendo o primeiro uma proposição lógica ou relacional, seguido de dois possíveis desfechos.

Em nosso exemplo, inicialmente será verificado se o valor atribuído a variável nota é igual ou maior que 7, em seguida é inserido o operador ternário “ ? “, seguido de dois possíveis desfechos, onde caso o valor de nota for maior ou igual a 7 será retornado ‘Aprovado’, caso contrário será retornado ‘Reprovado’.

Exibindo em tela via console.log( ) o último valor atribuído a variável media, o retorno será Aprovado.

ESTRUTURAS CONDICIONAIS

Uma vez entendidas as estruturas sintáticas básicas da linguagem JavaScript, podemos finalmente partir para a estruturas funcionais, que como próprio nome sugere, dão funcionalidades a nosso código, sendo a primeira delas a ser explorada as chamadas estruturas condicionais.

Raciocine que uma das estruturas de código mais comuns, independentemente do tipo de aplicação é uma estrutura voltada a resolver uma tomada de decisão com base em gatilhos acionados pelo usuário.

Também chamadas de controle de fluxo, tais estruturas basicamente inserem em nosso código pontos de tomada de decisão, alterando o comportamento do código a partir desse ponto conforme o usuário vai aderindo a certas opções.

Em outras palavras, certas funcionalidades de nosso programa só serão executadas conforme uma condição imposta for validada. Pela lógica, não existe como o usuário tomar uma decisão que não está condicionada ao código, logo, todas as possíveis tomadas de decisão devem ser programadas de forma lógica e responsiva, haja visto que toda ação em um programa gera uma reação.

Assim como tudo o que foi visto até então, uma estrutura condicional possui uma sintaxe própria e algumas regras básicas que devemos seguir para que o interpretador da linguagem consiga de fato tal estrutura como um controle de fluxo de execução.

if, else if, else

Por parte de nomenclatura, faremos uso de algumas palavras ao sistema como if, else if, else, switch e default para escrever as sentenças a serem validadas. E se tratando das expressões lógicas que estaremos criando, raciocine que basicamente estaremos programando uma determinada ação a ser executada, porém a mesma somente será executada se uma condição importa for interpretada como verdadeira.

Dessa forma, conseguimos fazer com que uma estrutura de código bastante simples controle o desfecho de determinadas ações do programa de acordo com as escolhas do usuário.

```
1   let nota = 85
2
3   if (nota >= 70) {
4       console.log('Aprovado!!!')
5   }
6
```

Partindo para o código, logo na primeira linha declaramos uma variável de nome nota, que recebe como atributo o valor 85.

Em seguida criamos a estrutura condicional, preste atenção à sintaxe, a expressão em si começa com a palavra reservada if, que em tradução livre significa “se”, seguido da expressão lógica entre parênteses, nesse caso, verificando se o valor de nota é igual ou maior que 70, onde caso seja, o bloco de código indentado entre chaves é executado, nesse caso, exibindo em tela a mensagem “Aprovado!!!’.

Essa é a forma mais básica de se criar uma estrutura condicional, usando if, seguido de que condição deverá ser atingida, seguido de o que fazer caso a condição tenha sido atingida.

Em nosso exemplo, se o valor de nota for igual ou maior que 70 é exibido em tela “Aprovado!!!”, caso contrário simplesmente esse bloco de código é ignorado.

```
let nota = 55

if (nota >= 70) {
    console.log('Aprovado!!!')
} else {
    console.log('Reprovado!!!')
}

```

No exemplo anterior, caso a condição imposta não fosse validada como True, nada acontecia, porém, o comum é em nossas estruturas condicionais programar dois ou mais desfechos, sendo um executado quando a condição é atingida, e outro executado quando a condição imposta não for atingida. Para esse contexto usamos também da palavra reservada ao sistema else.

Em nosso exemplo, por parte de sintaxe, logo após o bloco de código criado anteriormente é inserido o marcador else seguido de um novo bloco de código indentado entre chaves, nesse caso também exibindo em tela uma mensagem ao usuário.

Apenas para fins de exemplo, note que o valor da variável nota agora é 55, sendo assim, o interpretador irá ler a expressão escrita na linha 3 do código, e como a condição não é validade como True (pois o valor de nota agora não é nem igual nem maior que 70), o mesmo irá pular para a linha onde está o else, executando seu respectivo bloco de código.

Nesse caso, o retorno será: “Reprovado!!!”

```
let nota = 82

if (nota == 100) {
    console.log('Aprovado com nota máxima!!!')
} else if (nota >= 70 && nota <= 99) {
    console.log('Aprovado por média!!!')
} else {
    console.log('Reprovado!!!')
}

```

Avançando alguns passos, diferentemente dos exemplos anteriores onde de acordo com a condição imposta tínhamos ou um desfecho ou outro, uma prática comum é criarmos estruturas condicionais com mais de duas possibilidades, e isto é feito de forma simples através de else if.

Vamos ao exemplo, reutilizando o código anterior, vamos inserir uma estrutura condicional intermediária entre if e else, usando de else if. Tenha em mente que não existe nenhuma restrição quanto ao número de else ifs a serem usados, desde que os mesmos tenham um propósito claro em meio ao código.

Repare que else if também recebe uma expressão lógica a ser validada, com seu respectivo desfecho que é exibir em tela uma mensagem ao usuário.

Sendo assim, quando o interpretador do núcleo da linguagem lê a linha 3 do código, se depara com a estrutura condicional inicial, verificando se o valor de nota é igual a 100, como não é, ele pula para a linha 5 onde temos else if, então ele verifica neste ponto se o valor de nota é igual ou maior que 70 e igual ou menor que 99, nesse caso, como o valor de nota está dentro desse intervalo, então é executada alinha de código referente a else if e o processo é encerrado.

```
1   let nota = 100
2
3   if (nota == 100) {
4       console.log('Aprovado com nota máxima!!!')
5   } if (nota >= 70 || nota <= 99) {
6       console.log('Aprovado por média!!!')
7   } else {
8       console.log('Reprovado!!!')
9   }
10
```

Outra possibilidade interessante para alguns contextos é criar estruturas condicionais que permitam mais de uma opção como verdadeira.

Como visto anteriormente, comumente usamos if como o primeiro identificador de estrutura condicional, seguido de quantos else ifs forem necessários para todas as possibilidades intermediárias, e um else para quando nenhuma das condições é validada.

Porém, é perfeitamente possível criar estruturas condicionais com uma, duas ou mais possibilidades, simplesmente aninhando cada um dos possíveis desfechos usando como marcador if.

Em nosso exemplo, agora usando como valor para a variável nota o valor 100, note que temos duas expressões lógicas, em duas estruturas condicionais, que podem ser validadas como True.

A primeira delas logo na linha 3 do código, pois o retorno da expressão se o valor de nota é igual a 100 é True. A segunda delas sendo na linha 5, pois analisando a expressão condicional, a mesma é uma expressão lógica composta, onde se o valor de nota foi igual ou maior que 70 (True), ou se o valor de nota for igual ou menor que 99 (False), True e False = True.

Sendo assim, o interpretador identificou duas condicionais validadas como True, executando ambas sem nenhuma distinção.

switch, case

```
let nota = 8

switch (nota) {
    case 10:
        console.log('Aprovado com nota máxima!!!')
        break
    case 9:
    case 8:
    case 7:
        console.log('Aprovado por média!!!')
        break
    case 6:
    case 5:
    case 4:
    case 3:
    case 2:
    case 1:
        console.log('Reprovado!!!')
}

```

Ainda se tratando de estruturas condicionais / para controle de fluxo, uma estrutura muito comum a linguagens de programação interpretadas é o chamado switch-case.

Essa estrutura condicional por sua vez tem suas particularidades no que diz respeito tanto a sintaxe, quanto a comportamento, sendo útil para casos onde queremos criar uma estrutura de validação baseado em escolha direta pelo usuário.

Como exemplo, de início é declarada uma variável de nome nota que recebe como atributo o valor 8, apenas para fins de exemplo.

Na sequência criamos a estrutura switch, que recebe um dado/valor ou uma expressão lógica a ser validada, seguido de todos os possíveis desfechos de acordo com seu critério de controle.

Nesse caso, caso o valor de nota for 10, então é exibido em tela uma mensagem ao usuário, encerrando o processamento de switch por meio do comando break. Como sabemos que não é 10, então são lidas as outras possibilidades, onde caso o valor seja 9, ou 8, ou 7, é exibido uma mensagem ao usuário para essa condição.

Da mesma forma, caso até então o valor de nota não houvesse sido validado por nenhuma opção, iria percorrendo todas as programadas em switch até que não houvessem mais opções, nesse caso, não realizando nenhuma ação.

```
let nota

switch (nota) {
    case 10:
        console.log('Aprovado com nota máxima!!!')
        break
    case 9: case 8: case 7:
        console.log('Aprovado por média!!!')
        break
    case 6: case 5: case 4: case 3: case 2: case 1:
        console.log('Reprovado!!!')
        break
    default:
        console.log('Sem nota...')
}

```

Finalizando este tópico referente a estruturas condicionais, vamos ver uma última possibilidade, acrescentando a palavra reservada default em nossa estrutura switch.

Equivalente ao else das estruturas condicionais convencionais, dentro de nossa estrutura switch é possível criar um retorno padrão para quando nenhuma das opções é validada pela condição.

Em nosso exemplo, simulando esse comportamento incomum, inicialmente declaramos uma variável de nome nota, sem atribuir nenhum dado/valor para a mesma, o que é permitido, mas não recomendável em JavaScript.

Em seguida, formatamos nosso código apenas para tornar as opções case mais compactas, de modo que tal organização não interfere em nada na execução das mesmas.

Por fim é criada uma linha default, com uma mensagem indentada para a mesma.

Executando esse bloco de código o retorno será “Sem nota...” pois de fato a variável nota não possui valor nenhum, consequentemente nenhuma das opções programadas em switch podem ser validadas, sendo assim acionado o gatilho para o retorno padrão via default.

```
let nota = 0

if (nota > 0) {
    if (nota >= 1 && nota <= 6) {
        console.log('Reprovado!!!')
    } else if (nota >= 7 && nota <= 9) {
        console.log('Aprovado por média!!!')
    } else if (nota == 10) {
        console.log('Aprovado com nota máxima!!!')
    }
} else {
    console.log('Nota inválida')
}

```

Apenas realizando um adendo, uma possibilidade que acabou não sendo demonstrada em nenhum exemplo é o uso de estruturas condicionais dentro de estruturas condicionais, ou estruturas condicionais aninhadas conforme o jargão usual.

Desde que se respeite a sintaxe correta de cada elemento e que cada camada de estrutura condicional tenha uma boa justificativa, é perfeitamente possível aninhar quantas estruturas condicionais forem necessárias.

No exemplo, declarada a variável nota com seu valor atribuído 0, em seguida é criada uma estrutura condicional aninhada onde, se o

valor de nota for maior que 0, é executado o bloco de código referente a linha 4 que por sua vez é uma nova estrutura condicional, verificando dessa vez se o valor de nota é igual ou maior que 1 e se o valor de nota é igual ou menor que 6.

O ponto aqui é que, como nossa variável nota possui valor 0, a primeira camada da estrutura condicional, verificando se a mesma possuía valor atribuído maior que 0 já é validada como False, logo, todo seu bloco de código é ignorado, sendo diretamente executado o código referente a linha 11 do código.

Caso o valor de nota fosse qualquer valor maior que 0, iria passar pela primeira fase de validação, caindo na estrutura condicional que a verificaria de acordo com alguns intervalos, retornando suas respectivas mensagens.

ESTRUTURAS DE REPETIÇÃO

Avançando com nossos estudos, agora que sabemos criar estruturas condicionais começamos a expandir o leque de possibilidades do que fazer com nossas variáveis em meio a um código, logo, podemos avançar para as chamadas estruturas de repetição.

Sem querer ser, mas já sendo redundante, estruturas de repetição servem para repetir uma determinada ação por um número determinado (ou indeterminado) de vezes, ou até que uma certa condição seja atingida.

while

Por parte de sintaxe, usaremos da palavra reservada while, em tradução livre “enquanto”, para criar estruturas em blocos de código que serão executadas em loop de acordo com algum critério estabelecido pelo desenvolvedor.

```
let numero = 1

while (numero <= 8) {
    console.log(numero)
    numero += 1
}

```

Indo diretamente para o exemplo, de início declaramos uma variável de nome numero, que por sua vez recebe como atributo o valor 1.

Em seguida, criamos nossa estrutura de repetição while, seguido de uma expressão lógica a ser validada, com seu respectivo bloco de código indentado caso a condição imposta tenha sido atingida.

Em nosso exemplo definimos que, enquanto o valor da variável numero for menor ou igual a 8, será exibido em tela via console.log( ) o próprio número, em seguida a variável numero terá seu valor atualizado para seu número atual acrescido de uma unidade.

Note que a expressão em si a ser validada é uma simples expressão lógica, podendo ser qualquer tipo de expressão, simples ou composta, desde que quando validada retorne um valor booleano, assim como no corpo do laço de repetição é necessário criar o mecanismo que atualiza o valor da variável de referência, caso contrário, o bloco de código irá entrar em um loop infinito pois o mesmo nunca atingirá o gatilho final, nesse caso, ser igual a 8.

Executando esse bloco de código o retorno será:

1

2

3

4

5

6

7

8

E desse retorno é importante salientar que cada valor será exibido em uma linha, pois a cada linha temos um novo ciclo de execução, até que a variável tenha como atributo o valor 8, onde pelo qual já não será mais possível validar como True a expressão lógica definida para while.

```
let numero = 1

while (numero <= 8) {
    console.log(numero)
    numero += 1
    if (numero == 4) {
        break
    }
}

```

Se tratando de mecanismos criados para ter o comportamento de gatilho final, impedindo que nosso código entre em um loop infinito, podemos perfeitamente usar de estruturas condicionais dentro de estruturas de repetição. E por parte de estruturas condicionais, como vimos anteriormente, as mesmas podem acionar o gatilho break para que a execução do código seja interrompida a partir daquele ponto.

No exemplo, enquanto o valor de numero for menor ou igual a 8, o bloco indentado para while será executado, porém, de acordo com a condicional inserida no mesmo, se em algum momento o valor de número for igual a 4, então todo o processo é encerrado via break.

```
let numero = 1

do {
    console.log(numero)
    numero += 1
    } while (numero <= 5)

```

Ainda referente a while, uma observação importante a se fazer é que, como mencionado em outros momentos, a ordem da leitura léxica do interpretador para algumas sentenças de código é literal, logo, a ordem de escrita das estruturas é muito relevante para ditar o comportamento de um código.

Apenas como exemplo, uma possibilidade interessante é fazer com que um determinado bloco de código seja executado (ao menos uma vez) de acordo com um critério, que pode ser uma estrutura de repetição.

Em nosso código, declarada a variável numero com seu respectivo atributo, em seguida é criada uma estrutura do (palavra reservada, de tradução livre “faça”), com seu respectivo bloco de código, que será executado repetidas vezes de acordo com a regra definida em while para o mesmo.

Nesse caso, ao executar nosso código o retorno será:

1

2

3

4

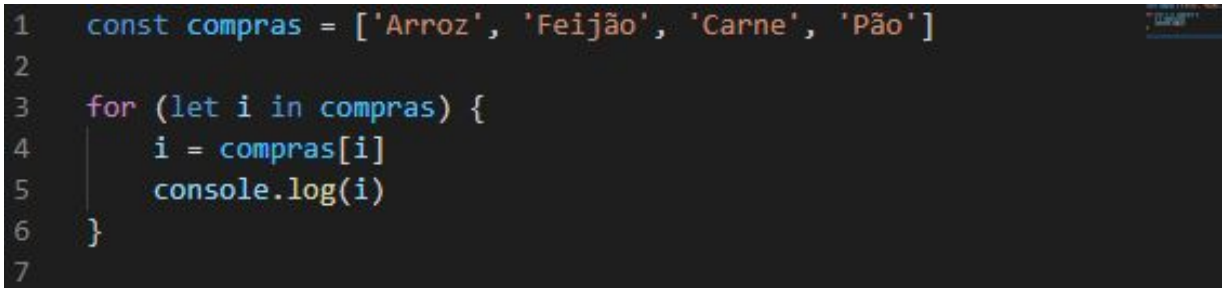
5

Por mais estranho que possa parecer neste momento, em JavaScript para certas estruturas de código é permitido o uso de uma expressão depois de um bloco, transformando a expressão em uma regra a ser seguida caso tal bloco seja reutilizado / executado repetidas vezes.

for

Das estruturas de repetição, outra estrutura muito útil da linguagem JavaScript é o laço for, que diferentemente de while, pode ser aplicado em situações onde precisamos percorrer todos elementos de um contêiner de dados (array, object, entre outros), iterando sobre cada um deles a cada ciclo de repetição.

O laço for por sua vez, na própria estrutura recebe uma variável temporária, que a cada ciclo de repetição receberá como atribuição um dado/valor extraído de um contêiner de dados, e dentro desse ciclo será utilizada para alguma função.



```
1  const compras = ['Arroz', 'Feijão', 'Carne', 'Pão']
2
3  for (let i in compras) {
4      i = compras[i]
5      console.log(i)
6  }
7
```

Como exemplo, vamos escrever um modelo básico de laço for.

Inicialmente é criada uma constante de nome compras, que recebe como atributo uma array contendo quatro elementos em forma de string. Relembrando que para o tipo de dados array, o mesmo é identificado por dados (independentemente de seu tipo, desde que respeitada sua sintaxe) entre colchetes e separados por vírgula.

Na sequência criamos um laço for, que usando da expressão let i in compras, a cada ciclo está criando uma variável de nome i, que recebe um dos elementos de dentro da variável compras.

Então é criado o bloco de código referente a ação que será tomada a cada ciclo de execução usando do último valor atribuído para a variável temporária i, nesse caso, instanciando a variável i e atribuindo para a mesma o último dado/valor associado a variável temporária i, em seguida simplesmente exibindo em tela seu conteúdo através da função console.log( ).

A cada ciclo de execução a variável temporária i terá um dado/valor associado (sendo no primeiro ciclo o elemento de posição de índice 0 da array, por segundo o elemento de índice número 1, e assim por diante até que seja feita a iteração sobre todos elementos da array) que será usado pelo bloco de ação do laço for.

Executando esse bloco de código o retorno será:

Arroz

Feijão

Carne

Pão

```
1   const compras = ['Arroz', 'Feijão', 'Carne', 'Pão']
2
3   for (let i = 0; i < compras.length; i++) {
4       console.log(i)
5   }
6
```

Trazendo um outro exemplo, para assim ver outras possibilidades, note que agora a expressão escrita como regra para o laço for envolve além da variável temporária i, uma expressão lógica e um incremento sobre a variável i.

Abstraindo a expressão, inicialmente é declarada uma variável de nome i, com valor inicial atribuído igual a zero, enquanto i for menor que o tamanho total da array compras, o valor de i é atualizado, tendo seu valor acrescido em uma unidade a cada ciclo de repetição.

Dessa forma, percorremos todos os elementos da array compras, e com base no número retornado, podemos saber seu tamanho / número de elementos, algo útil para contextos onde estamos um portando uma array a qual não sabemos seu conteúdo.

Voltando ao exemplo, executando seu código serão retornados os valores 0, 1, 2 e 3, sendo assim, sabemos que compras é uma array composta de 4 elementos.

```
const compras = ['Arroz', 'Feijão', 'Carne', 'Pão']

for (let i = 0; i < compras.length; i++) {
    console.log(compras[i])
}

```

Exibindo em tela, a cada ciclo de execução, o conteúdo da variável compras na posição i, chegamos no mesmo resultado que obtivemos quando usada a expressão i in compras. O ponto é que em JavaScript existem diversas maneiras de se chegar ao mesmo resultado, logo, fica totalmente a critério do desenvolvedor escolher qual forma o mesmo julga mais eficiente.

```
const funcionario1 = {
    Nome: 'Milena',
    Sobrenome: 'Silva',
    Função: 'Enfermeira'
}

for (func in funcionario1) {
    console.log(`${func} - ${funcionario1[func]}`)
}

```

Por fim, por hora encerrando este capítulo introdutório ao laço for, outra possibilidade a ser demonstrada é que o mesmo tem plena capacidade de percorrer um object, extraindo dados / valores do mesmo, desde que se respeite sua sintaxe de elementos dispostos em campos de chave: valor.

Como exemplo, é criada uma constante de nome funcionario1, que por sua vez recebe como atributo um object composto de três conjuntos de chave:valor. Como mencionado anteriormente, para este tipo de dado em JavaScript, o convencional é fazer referência a uma determinada chave, para assim acessar seu valor.

Porém, usando do laço for podemos percorrer todos elementos deste object, extraindo os dados que bem quisermos do mesmo.

Por exemplo, criando um laço for que, a cada ciclo de execução retornará um dado para a variável temporária func (lembrando que o nome da variável temporária pode ser qualquer nome, exceto palavras reservadas ao sistema), sendo este dado o valor atribuído a sua chave.

Exibindo em tela via console.log( ) o conteúdo de func, assim como de funcionario1 na posição func, o retorno será:

Nome – Milena

Sobrenome – Silva

Função – Enfermeira

*No exemplo foi usado dentro de console.log( ) uma interpolação, usando de máscaras de substituição que inserem dados de variáveis diretamente sobre os campos delimitados com chaves, algo que veremos em detalhes em capítulos subsequentes.

FUNÇÕES

Ao longo dos capítulos anteriores, para praticamente todos os exemplos apresentados acabamos por usar da função console.log( ), uma função básica de retorno, que exibe em tela (via terminal) algum retorno referente a execução de nosso código.

Agora que já temos uma certa bagagem de conhecimento acumulada, podemos nos aprofundar no que diz respeito a funções na linguagem JavaScript, entendendo suas particularidades e criando nossas próprias funções.

Uma função, independentemente da linguagem de programação, nada mais é do que um bloco de código que pode ser executado e reutilizado livremente, quantas vezes for necessário, de modo a realizar ações personalizadas de nosso programa. Em JavaScript, uma particularidade a destacar é que por parte de leitura léxica, o interpretador ao carregar um código realiza uma leitura e pré-carregamento de todas funções presentes no código, aplicando a leitura léxica sequencial apenas para os demais objetos.

Como mencionado em outros momentos, tudo em JavaScript é função, de modo que uma função é um tipo de dado que terá suas particularidades sintáticas e de comportamento. Na prática o que isso reflete é que em JavaScript usar de funções é algo bastante dinâmico, indo muito além de ser um script / algoritmo sendo executado para gerar um retorno, podendo oferecer inúmeras possibilidades à medida que entendemos suas estruturas internas com seus respectivos comportamentos.

Lembrando que JavaScript é uma linguagem multiparadigma, ou seja, por meio da mesma é possível implementar soluções de modo funcional, orientado a objetos, procedural, distribuído, etc... e tudo isso internamente é perfeitamente funcional graças ao modelo adotado onde tudo é função.

Sintaxe básica de uma função

```
1   function minha_funcao() {}
2
```

Para criar uma função personalizada, de forma literal, é necessário seguir a sintaxe acima. Usando da palavra reservada function seguido do nome da função (qualquer nome exceto palavras reservadas ao sistema), que em anexo, demarcado por parênteses tem um campo de inserção de parâmetros, por fim um campo para inserção de todos os blocos necessários para tal função.

Em JavaScript não existe a obrigatoriedade de se repassar parâmetros quando se declara uma função, assim como uma função não obrigatoriamente precisa retornar alguma coisa. Porém, sempre que criados parâmetros para uma função os mesmos deverão ser utilizados em algum momento pelas estruturas de código situadas no corpo da função, assim como o retorno em si de uma função é explícito quando as estruturas de dentro da função geram algum novo dado para alguma variável, ou implícita (retornando undefined) quando a função é terminada sem produzir nenhum dado.

No exemplo, é criada a função minha_funcao( ), que não realiza nenhuma ação nem retorna nenhum dado.

```
1   function minha_funcao() {}
2
3   const variavel1 = minha_funcao()
4
```

Uma prática muito comum é usar de uma função como atributo de uma variável / constante. Desse modo, podemos tanto chamar diretamente a função por seu nome funcional quanto pela variável que a instancia e inicializa.

O interessante de se atribuir uma função para uma determinada variável é que caso a função retorne algum dado / valor, o mesmo ficará atribuído a respectiva variável.

```
1  function mensagem1() {
2      console.log('Bem vindo(a)!!!')
3  }
4
5  let boas_vindas = mensagem1()
6
```

Partindo para algo funcional, uma função pode ou não conter parâmetros, e independentemente disso, executar alguma determinada ação quando for chamada.

Em nosso exemplo é criada a função mensagem1( ), sem parâmetros mesmo, que dentro de seu corpo possui a função console.log( ) parametrizada com uma frase em forma de string.

Na linha 5 do código temos uma variável de nome boas_vindas que chama a função mensagem1( ). Nesse caso, quando o interpretador da linguagem ler a linha 7 do código, será executada a função mensagem1( ), exibindo em tela sua respectiva mensagem.

```
1  function soma_2_num(num1, num2) {
2      console.log(num1 + num2)
3  }
4
5  let variavel1 = soma_2_num(11, 52)
6
```

Como dito anteriormente, uma função pode ou não gerar algum retorno, e nesse contexto, entenda que retornar algo é de fato produzir um dado / valor que poderá ser acessível e utilizável posteriormente por outras estruturas de código.

Quando criamos uma função, que explicitamente não retorna nenhum dado, a mesma até pode ser executada, porém imediatamente após sua execução tal dado é perdido, descarregado da memória.

Por exemplo, criada nossa função soma_2_num( ), que recebe dois parâmetros num1 e num2, respectivamente, note que dentro do

corpo da função temos uma função console.log( ) que exibirá em tela o resultado da soma entre num1 e num2.

De volta ao escopo global do código, é criada uma variável de nome variavel1 que chama a função soma_2_num( ) parametrizando a mesma com 11 e 52, respectivamente.

```
function soma_2_num(num1, num2) {
    console.log(num1 + num2)
}

let variavel1 = soma_2_num(11, 52)

console.log(variavel1)

```

Executando esse código será exibido o retorno da função console.log( ), nesse caso 63, porém se tentarmos buscar algum dado/valor atribuído a variável variavel1 teremos como resultado undefined, e isto se dá pela forma a qual escrevemos a estrutura de código de nossa função.

Apenas realizando uma pequena observação, uma vez que a função soma_2_num( ) recebe dois parâmetros, num1 e num2, respectivamente, quando uma variável / constante chama tal função é obrigatório que repasse para a mesma dados / valores para tais parâmetros, caso contrário será gerado um erro.

```
function soma_2_num(num1, num2) {
    return (num1 + num2)
}

let variavel1 = soma_2_num(11, 52)

console.log(variavel1)

console.log(variavel1 + 10)

```

Contornando essa situação, podemos simplesmente inserir dentro do corpo de nossa função o marcador return, palavra reservada ao

sistema que irá retornar um dado / valor referente à execução da função.

Realizando exatamente o mesmo processo do exemplo anterior, agora, em nossa linha 9 do código, ao buscar um dado / valor atribuído a variável variavel1 nos será retornado 63.

Uma vez que a variável variavel1 possui um atributo, podemos livremente iterar com o mesmo. Apenas para fins de exemplo, em nossa linha 9 do código, usando como parâmetro da função console.log( ) variavel1 + 10, nos é retornado 73, pois a soma do valor de variavel1 + 10 resulta neste valor.

O ponto a se destacar é, para certos contextos, é interessante que uma função gere algum retorno para alguma variável, e a forma como sintaticamente devemos organizar as estruturas de uma função molda seu comportamento.

Função com parâmetros definidos em justaposição

Complementar o entendimento dos fundamentos de funções em JavaScript, é importante salientar uma particularidade chamada justaposição.

De modo geral, uma vez que estamos a declarar parâmetros para nossas funções, o número de parâmetros e a ordem como os mesmos serão usados importa para o interpretador da linguagem.

```
1  function minha_funcao(n1, n2) {
2      let soma = n1 + n2
3      return soma
4  }
5
6  console.log(minha_funcao(1, 2))
7
```

Em nosso exemplo, note que a função minha_funcao( ) recebe obrigatoriamente dois parâmetros, n1 e n2 respectivamente, de forma que quando estivermos chamando tal função, o primeiro valor

declarado será repassado para o parâmetro n1, assim como o segundo parâmetro declarado será repassado para o parâmetro n1.

Esta simples regra é comumente chamada de justaposição.

```
1  function minha_funcao(n1, n2) {
2      let soma = n1 + n2
3      return soma
4  }
5
6  console.log(minha_funcao(1, 2, 3, 4, 5, 6))
7
```

Uma particularidade interessante da linguagem JavaScript, que normalmente em outras linguagens acabaria por gerar algum tipo de erro, é o fato de que, uma vez que temos uma função com número de parâmetros definido, caso ao se instanciar e inicializar tal função sejam repassados mais parâmetros do que a mesma espera, graças a justaposição o interpretador pegará os valores que alimentarão os parâmetros obrigatórios da função e simplesmente desconsiderará todos os demais.

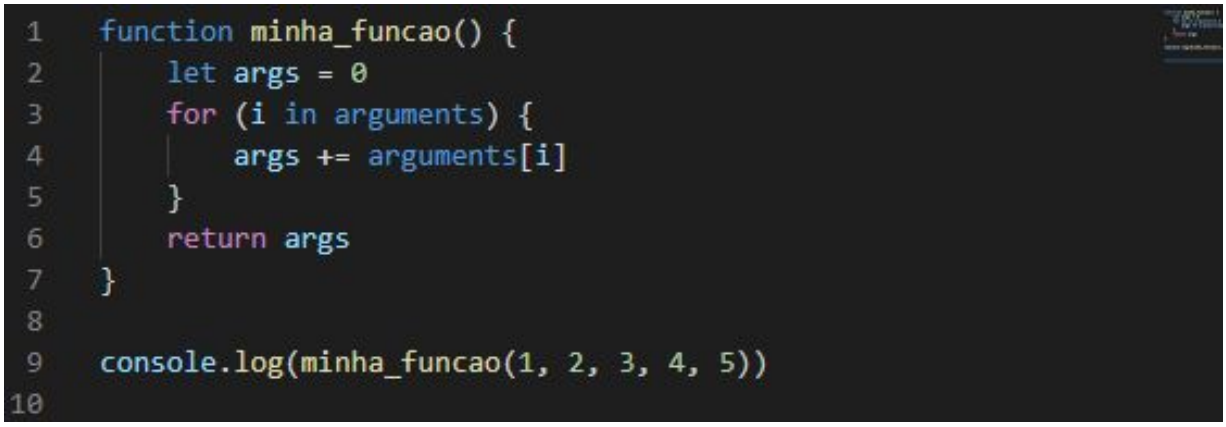
Em nosso exemplo, na linha 6 de nosso código estamos a parametrizar console.log( ) com nossa função minha_funcao( ) que nesse caso recebe como parâmetros os valores 1, 2, 3, 4, 5 e 6.

Nesse caso, o valor 1 será repassado para o parâmetro n1, assim como o valor 2 será repassado para o parâmetro 2 de minha_funcao( ).

Executando esse código o resultado será, como esperado, 3.

Função com parâmetros indefinidos

Para certos contextos, pode ocorrer de no momento da criação de uma determinada função não sabermos ao certo quantos ou quais parâmetros nossa função receberá, e isso em JavaScript pode ser contornado de forma simples.



```
function minha_funcao() {
    let args = 0
    for (i in arguments) {
        args += arguments[i]
    }
    return args
}

console.log(minha_funcao(1, 2, 3, 4, 5))
```

O método convencional, que recomendo fortemente a adoção, pois serve para se trabalhar com todo e qualquer tipo e número de dados, é fazendo o uso de arguments.

O objeto arguments, palavra reservada ao sistema, é uma espécie de marcador que suporta a inserção de múltiplos parâmetros indefinidos para uma função.

Em nosso exemplo, criada a função minha_funcao( ) note que a mesma não possui explicitamente nenhum parâmetro declarado. Já no corpo da função é criada uma variável de nome args que inicialmente tem como seu valor atribuído 0.

Em seguida é criado um laço for que percorrerá cada elemento repassado para arguments, que por sua vez recebe dados / valores transformando-os em parâmetros da função, a cada ciclo de execução atribuindo o valor lido para a variável temporária i.

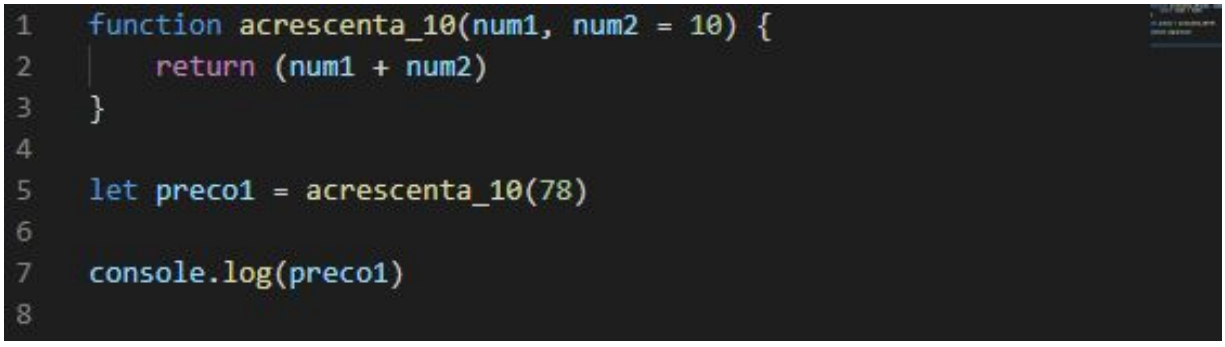
Por fim, como em nosso exemplo estamos simplesmente a somar cada valor repassado como parâmetro, a variável args é novamente instanciada, tendo seu valor atualizado, recebendo como atributo a

soma de seu valor atual com o último valor lido para a variável i. Encerrando a função é retornado o valor de args.

Fora da função, apenas para fins de exemplo, através da função console.log( ), repassando como parâmetro para a mesma a função minha_funcao( ), por sua vez parametrizada com os valores 1, 2, 3, 4 e 5, ao executar nosso código será retornado o valor 15, referente a soma dos números usados como parâmetros.

Função com parâmetro padrão

Outra possibilidade muito útil na prática é a de se criar os campos de parâmetros de nossas funções já com algum tipo de dado / valor pré-definido, de modo que caso o usuário não repasse nenhum dado / valor para tal parâmetro, será usado seu valor padrão, assim como caso o usuário repasse um dado / valor, o mesmo irá sobrescrever o dado / valor padrão.



```
1  function acrescenta_10(num1, num2 = 10) {
2      return (num1 + num2)
3  }
4
5  let preco1 = acrescenta_10(78)
6
7  console.log(preco1)
8
```

Diretamente ao exemplo, repare que logo na primeira linha criamos uma função de nome acrescenta_10( ) que por sua vez receberá um valor a ser usado como parâmetro para num1, e num2 nesse caso já possui um valor atribuído.

Dentro do corpo da função temos um simples comando para retornar a soma entre num1 e num2.

De volta ao escopo global do código, é criada uma variável de nome preco1, que instancia e inicializa a função acrescenta_10( ) parametrizando a mesma com o valor 78, que nesse caso será repassado em justaposição para num1.

Por fim, via console.log( ) é exibido em tela o conteúdo da variável preco1, nesse caso, 88.

Função arrow

Quando estamos aprendendo uma linguagem de programação, independentemente de qual, é perfeitamente normal criarmos algumas estruturas de código mal otimizadas, seja por sua legibilidade, seja por outros fatores que acarretem em má performance de execução do código.

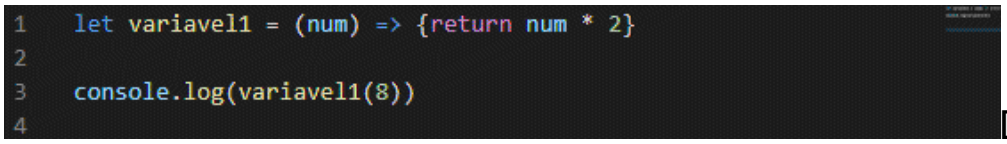
Conforme avançamos em nossos estudos, à medida que dominamos certos pilares fundamentais da linguagem, é natural que iremos conseguir aperfeiçoar nossos códigos por parte de sua estruturação.

Nessa linha de raciocínio, em JavaScript sempre temos uma vasta gama de possibilidades no que diz respeito a meios e métodos para se chegar em um resultado comum.

Uma prática muito comum entre desenvolvedores experientes é tentar reduzir seus códigos ao máximo, tornando-os não somente mais enxutos, mas mais eficientes de modo geral.

Toda e qualquer linguagem de programação tem seus meios de redução de código, com JavaScript não seria diferente, e se tratando de reduzir estruturas de funções, um dos métodos mais úteis é o chamado redução por função anônima / vazia.

Uma função anônima, como o próprio nome sugere, é uma função sem nome declarado, que pode ser escrita ao longo do código ou atribuída a uma variável, fazendo uso de uma notação um pouco diferente do usual, pois em JavaScript temos um operador específico para este tipo de função, que a identificará ao interpretador como uma função anônima.



Direto ao ponto, preste muita atenção à sintaxe. Inicialmente é declarada uma variável de nome variavel1, que recebe como atributo uma função

anônima, que por sua vez recebe como parâmetro um número qualquer e multiplica o mesmo por 2.

Repare que logo após o nome da variável com seu respectivo operador de atribuição “ = “ é criado um campo delimitado por parênteses com um objeto num dentro do mesmo, este campo é o campo de inserção de parâmetros para a função, que nesse caso, nem nome possui.

Na sequência é inserido o operador “ => “ arrow function, nesse caso, apontando para o campo onde delimitado por chaves está o corpo da função, que nesse caso simplesmente retorna o valor repassado para num multiplicado por 2.

Uma vez criada nossa função anônima, podemos usar da mesma diretamente como parâmetro para a função console.log( ). De acordo com nosso exemplo, parametrizando console.log( ) com variavel1(8) estamos chamando a função anônima através da variável variavel1, repassando como parâmetro para a mesma o valor 8.

Nesse caso, executando esse código o resultado será 16.

Função anônima (método convencional)

```
1   const soma = function (num1, num2) {return num1 + num2}
2
3   console.log(soma(11, 5))
4
```

Apenas demonstrando outra possibilidade, sem usar da notação de arrow function, podemos perfeitamente criar uma função “anônima” simplesmente não atribuindo um nome para a mesma.

No exemplo, criada uma constante de nome soma, note que a mesma recebe como atributo uma função (de acordo com a palavra reservada function) com seus respectivos campos para parâmetros e para o corpo da função.

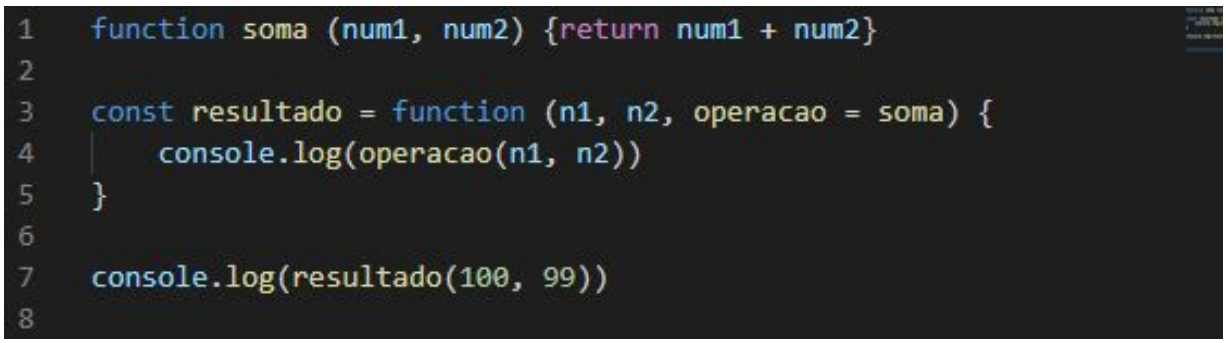
Usando a constante soma como parâmetro para console.log( ), com seus respectivos parâmetros, executando o código a função em si será executada normalmente.

O ponto a destacar, quando falamos de funções reduzidas em forma de funções anônimas, é que justamente podemos usar de suas particularidades para em meio a um código escrever funções bastante específicas, especializadas em uma única tarefa que a nível de código estarão condensadas.

Funções aninhadas

Agora que entendemos o básico sobre os meios e métodos para criarmos nossas próprias funções, podemos aplicar um pouco de tudo o que foi visto até então explorando outras possibilidades.

Uma das possibilidades interessantes em JavaScript é que, se tratando de reutilização de blocos de código, como tudo em JavaScript é uma função, nada impede que façamos uso de funções dentro de funções, funções que trabalham em conjunto (aninhadas), funções que são executadas a partir do retorno de outras funções, etc... E para isto vamos direto para o exemplo:



```
1  function soma (num1, num2) {return num1 + num2}
2
3  const resultado = function (n1, n2, operacao = soma) {
4      console.log(operacao(n1, n2))
5  }
6
7  console.log(resultado(100, 99))
8
```

De início, criamos uma função via método convencional de nome soma( ), que recebe dois parâmetros num1 e num2, retornando o resultado da soma entre tais valores.

Na sequência é criada uma constante de nome resultado que por sua vez é parametrizada com uma função anônima que recebe três parâmetros em justaposição, dois a serem repassados pelo usuário e um terceiro parâmetro nomeado de operação que recebe como valor padrão a função soma( ) escrita anteriormente.

Como retorno desta função é exibido em tela via console.log( ) o resultado da soma dos valores usados como parâmetros para a função soma( ).

Por fim, diretamente como parâmetro de console.log( ) repassamos a constante resultado com valores 100 e 99 a serem repassados como parâmetros de sua função anônima.

Em outras palavras, note que aqui uma função chama a outra, que chama outra, criando uma cadeia hierárquica onde a função soma( ) está na camada mais interna, sendo instanciada dentro da função anônima atribuída a resultado, de modo que podemos através da constante resultado fazer uso da função soma( ) sempre que necessário.

Executando esse bloco de código, o retorno será 199.

MÉTODOS APLICADOS

Uma vez que entendemos a construção dos principais tipos de dados da linguagem JavaScript, podemos buscar entender os principais métodos que se aplicam a tais tipos de dados, no sentido de fazer uso de funções pré-moldadas disponíveis no núcleo da linguagem.

Toda e qualquer linguagem de programação conta com um acervo de funções prontas, pré-configuradas de modo a facilitar a aplicação de certas funções sobre certos objetos, descartando a necessidade de se programar do zero funções básicas.

Para tornar este capítulo mais dinâmico, vamos criar uma estrutura básica de cada tipo de dado, manipulando o mesmo através de funções embutidas da linguagem.

Métodos aplicados a strings

Se tratando dos meios e métodos de manipulação de strings, a linguagem JavaScript oferece uma série de ferramentas de fácil implementação que nos permite manipular nossas strings de forma fácil e objetiva.

```
1   let mensagem1 = 'Bem vindo(a) a Curitiba'
2
3   console.log(mensagem1[0])
4
```

Lembrando que uma string é um tipo de dado de conteúdo textual, onde cada elemento dessa string possui um número interno de índice, sendo assim possível instanciar um ou mais elementos de uma string através de sua posição de índice.

Como exemplo, declarada a variável mensagem1 que recebe a string ‘Bem vindo(a) a Curitiba’, podemos iterar individualmente sobre cada um dos elementos que compõem essa frase.

Repassando como parâmetro para console.log( ) nossa variável mensagem1 com o marcador [ ], estaremos fazendo a referência a uma determinada posição de índice, nesse caso, o elemento de posição de índice 0 é a letra B.

replace( )

```
1   let mensagem1 = 'Bem vindo(a) a Curitiba'
2
3   console.log(mensagem1)
4
5   mensagem1 = mensagem1.replace('Curitiba', 'Porto Alegre')
6
7   console.log(mensagem1)
8
```

Dando início às devidas explicações sobre métodos embutidos aplicados, vamos começar praticando algumas possíveis manipulações de dados a partir de strings.

Em nosso exemplo, é declarada uma variável de nome mensagem1 que recebe como atributo a frase ‘Bem vindo(a) a Curitiba’ em forma de string.

Lembrando que uma string suporta todo e qualquer tipo de caractere alfanumérico, incluindo espaços em branco e quantas palavras forem necessárias.

Através de console.log( ) parametrizado com mensagem1, quando executado este bloco de código nos é retornado, como esperado, Bem vindo(a) a Curitiba.

Para realizar a substituição de um elemento de uma string, podemos aplicar o método replace( ), atualizando o conteúdo da própria variável.

O método replace( ) por sua vez recebe basicamente dois parâmetros em justaposição, sendo o primeiro deles a “palavra” a ser substituída, seguido da nova “palavra”

Nesse caso, na linha 5 do código instanciamos novamente a variável mensagem1, que recebe como atributo a instância dela própria, sendo aplicado o método replace( ) sobre a mesma, por sua vez parametrizado com ‘Curitiba’, ‘Porto Alegre’.

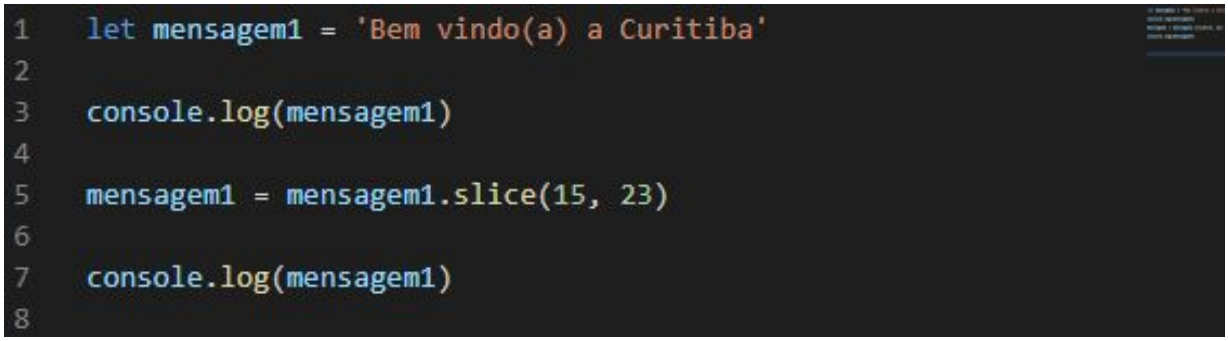
Exibindo em tela o conteúdo de mensagem1 via console.log( ) será possível ver que de fato os devidos elementos foram substituídos, compondo uma nova frase Bem vindo(a) a Porto Alegre.

slice( )

Uma prática comum quando estamos a manipular uma string é realizar o chamado fatiamento da mesma, extraindo um

determinado segmento de seu texto através dos números de índice onde tais elementos se encontram.

Lembrando que tanto strings quanto listas são objetos internamente mapeados e indexados, de modo que o índice começa sempre pelo elemento 0 e que em uma string cada caractere (incluindo espaço em branco) é um elemento de índice único.



```
1  let mensagem1 = 'Bem vindo(a) a Curitiba'
2
3  console.log(mensagem1)
4
5  mensagem1 = mensagem1.slice(15, 23)
6
7  console.log(mensagem1)
8
```

Declarada a variável mensagem1, com sua respectiva string, note que na linha 5 de nosso código estamos a atualizar o dado/valor de mensagem1, extraindo e atribuindo para a mesma via slice( ) apenas os elementos entre a posição de índice 15 até 22.

Exibindo em tela o conteúdo de mensagem1 via console.log( ) nos é retornado Curitiba.

*Para usar de números de posição de índice contados do fim para o começo da string, basta que tais valores estejam representados como índice negativo, por exemplo -1 até -12.

toUpperCase( )

Em situações onde se faz necessário converter todos os caracteres de uma string para letras maiúsculas, basta simplesmente aplicar o método toUpperCase( ), sem parâmetros mesmo, sobre a variável a qual a string está atribuída.

```
1   let mensagem1 = 'Bem vindo(a) a Curitiba'
2
3   mensagem1 = mensagem1.toUpperCase()
4
5   console.log(mensagem1)
6
```

Na linha 3 de nosso código, é aplicado sobre mensagem1 o método toUpperCase( ), atualizando o conteúdo de mensagem1.

Exibindo em tela via console.log( ) o conteúdo de mensagem1 será retornado BEM VINDO(A) A CURITIBA.

toLowerCase( )

O processo inverso também pode perfeitamente ser realizado, convertendo todos os elementos de uma string para letras minúsculas através de toLowerCase( ).

```
1   let mensagem1 = 'Bem vindo(a) a Curitiba'
2
3   mensagem1 = mensagem1.toLowerCase()
4
5   console.log(mensagem1)
6
```

Aplicado o método toLowerCase( ) sobre mensagem1, em seguida exibindo seu conteúdo atualizado em tela via console.log( ), o retorno será bem vindo(a) a Curitiba.

concat( )

Quando queremos realizar a união de duas strings, podemos tanto usar do operador de soma + quanto do método concat( ), sendo que ambos os métodos suportam unir diretamente as strings ou unir as mesmas a partir de variáveis.

```
let mensagem1 = 'Bem vindo(a) a '
let mensagem2 = 'Curitiba'

let mensagem = mensagem1.concat(mensagem2)

console.log(mensagem)

```

No exemplo temos inicialmente duas variáveis declaradas de nomes mensagem1 e mensagem2, respectivamente, seguido de uma nova variável de nome mensagem, que como atributo recebe a concatenação dos conteúdos das variáveis mensagem1 e mensagem2 de acordo com a sentença, fazendo uso do método concat( ).

Nesse caso, exibindo em tela através de console.log( ) o conteúdo de mensagem o retorno será Bem vindo(a) a Curitiba.

trim( )

Em situações onde uma string é composta de espaços em branco desnecessários antes e depois da “frase” em si, aplicar o método trim( ) sobre a mesma irá remover todos os espaços desnecessários.

```
let mensagem1 = '            Bem vindo(a) a Curitiba    '

mensagem1 = mensagem1.trim()

console.log(mensagem1)

```

A nível de exemplo, simulamos uma string que poderia ser importada de algum lugar, perdendo sua formatação e inserindo uma série de espaços desnecessários.

Atualizando o conteúdo da variável mensagem1, simplesmente aplicando sobre a mesma o método trim( ), todos os espaços antes

e depois dos elementos válidos da string serão removidos.

Novamente exibindo em tela o conteúdo de mensagem1 via console.log( ), o retorno será Bem vindo(a) a Curitiba.

split( )

Outra possibilidade interessante é a de dividir uma string em duas ou mais partes por meio de um critério estabelecido, normalmente um símbolo a ser identificado na string para que o interpretador identifique e divida a sentença exatamente a partir daquele ponto.

```
1   let mensagem1 = 'Bem vindo(a), a Curitiba'
2
3   mensagem1 = mensagem1.split(",")
4
5   console.log(mensagem1)
6
```

Como exemplo, usando da mesma variável e string dos exemplos anteriores, repare que na linha 3 estamos a atualizar o conteúdo da variável mensagem1, aplicando sobre a mesma o método split( ) por sua vez parametrizado com “,”, para nesse caso, o interpretador usar como referência para a separação o(s) local(is) onde houverem vírgula.

O comportamento do retorno dessa função é um pouco diferente do convencional, pois, uma vez que uma string é separada em partes via split( ), as partes separadas se tornam elementos de uma array.

Em nosso exemplo, exibindo em tela o conteúdo de mensagem1 através da função console.log( ), o retorno será [‘Bem vindo(a)’, ‘ a Curitiba’].

```
3   mensagem1 = mensagem1.split(",")
4
5   console.log(mensagem1[0])
6
```

Em função dessa conversão de tipo de dado, haja visto que agora o conteúdo de mensagem1 é uma array, ao buscar novamente sua posição de índice 0 agora o elemento de índice 0 é toda a string ‘Bem vindo(a)’.

Métodos aplicados a números

Se tratando de números, o convencional é que para os mesmos sejam aplicadas funções matemáticas, porém, para certos contextos específicos, a aplicação de certos métodos pode tornar as coisas mais funcionais em nossos códigos.

toFixed( )

```
1   let valor = 15.593928048923
2
3   console.log(valor.toFixed(2))
4
```

Dependendo do tipo de operação aritmética a qual estivermos trabalhando, é comum a produção de resultados extensos representados por números com casas decimais.

Usando do método toFixed( ) podemos facilmente definir quantos números devem ser considerados após a vírgula de um valor decimal.

No exemplo, declarada uma variável de nome valor, que possui como atributo um número bastante extenso, aplicando sobre a variável o método toFixed( ) podemos facilmente definir quantas casas decimais serão visíveis.

Nesse caso, exibindo em tela o conteúdo de valor via console.log( ), aplicando sobre a mesma o método toFixed( ) por sua vez parametrizado com o número de casas decimais, nos será retornado o valor esperado, nesse caso, 15.59.

Number( )

Das particularidades sintáticas das linguagens de programação, vimos anteriormente que em JavaScript toda sequência de caracteres (incluindo números) representada entre aspas é reconhecida como do tipo string.

Dessa forma, se por exemplo atribuirmos a uma variável o dado/valor 12345 o mesmo será identificado como do tipo number, enquanto ‘12345’ será reconhecido como string graças a sua notação.

Para certos contextos, é possível converter tipos de dados por meio de funções, nesse caso, converter outro tipo de dado para numérico através de eu método construtor Number( ).

```
1   let valor = '19.90'
2
3   valor = Number(valor)
4
5   console.log(typeof(valor))
6
```

Declarada uma variável de nome valor, note que como atributo para a mesma está o valor 19.90 representado como string.

Atualizando o valor da variável valor, aplicando sobre a mesma o método Number( ), a partir deste ponto a mesma variável com seu respectivo valor passarão a ser identificados pelo interpretador como objeto de tipo numérico.

Exibindo em tela o tipo da variável valor através de nossa função console.log( ), parametrizada com typeof( ), o retorno será number.

parseInt( ) e parseFloat( )

De forma um pouco diferente do exemplo anterior, realizar a conversão de um tipo de dado qualquer usando dos métodos parseInt( ) e parseFloat( ) realiza as devidas conversões apenas

sobre elementos compatíveis, ou seja, já numéricos em outro tipo de dado.

Raciocine, por exemplo, uma string ‘Ano 2001’ atribuída a uma variável. Convertendo a mesma via parseInt( ), nos será retornado apenas o valor 2001, desconsiderando todos os demais caracteres não numéricos.

O mesmo se aplica para casos onde temos números de casas decimais, uma vez que estamos convertendo um determinado valor para numérico de tipo inteiro, as casas decimais serão desconsideradas.

```
let valor = '19.90'

valor = parseInt(valor)

console.log(typeof(valor))

console.log(valor)

```

Para fins de exemplo, usando da mesma variável do exemplo anterior, agora atualizando o conteúdo da mesma através do método parseInt( ), seu conteúdo não somente será alterado de tipo (string para number) como de tipo numérico (número decimal para inteiro).

Usando novamente de console.log( ), exibindo em tela o tipo de dado de valor será retornado number. Exibindo em tela o valor da variável valor, será retornado 19.

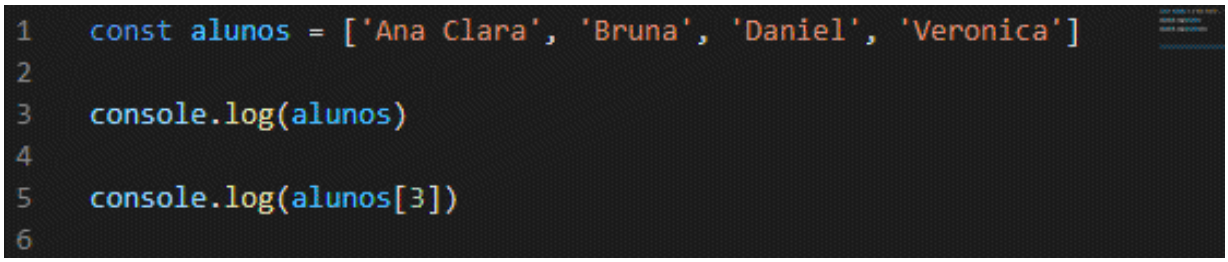
```
let valor = '19.90'

valor = parseFloat(valor)

console.log(typeof(valor))

console.log(valor)

```

Aplicando o método parseFloat( ) para atualizar o conteúdo da variável valor, exibindo seu conteúdo em tela via console.log( ) será possível notar o valor 19.9, desconsiderando o zero por ser um valor nulo.

Métodos aplicados a arrays

Equivalente as listas de outras linguagens de programação, uma array nada mais é do que um contêiner de dados onde é possível armazenar múltiplos dados independentemente de seu tipo, de modo que possamos iterar sobre tais dados como um todo ou individualmente, desde que respeitemos sua sintaxe e posição de índice.

Arrays, no que diz respeito a suas funcionalidades, não somente é útil por sua capacidade de armazenamento, mas é útil graças a uma série de métodos que nos permite manipular seus elementos internos de acordo com nossas necessidades.



```
1   const alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']
2
3   console.log(alunos)
4
5   console.log(alunos[3])
6
```

Partindo para a prática, logo de início declaramos uma constante de nome alunos, que por sua vez recebe em formato de array uma lista com 4 nomes.

Exibindo em tela via console.log( ) o conteúdo da constante alunos nos será retornado a própria lista de elementos, enquanto exibindo em tela o conteúdo de tal constante usando da notação de posição de índice, nos será retornado apenas o elemento referente a tal posição de índice.

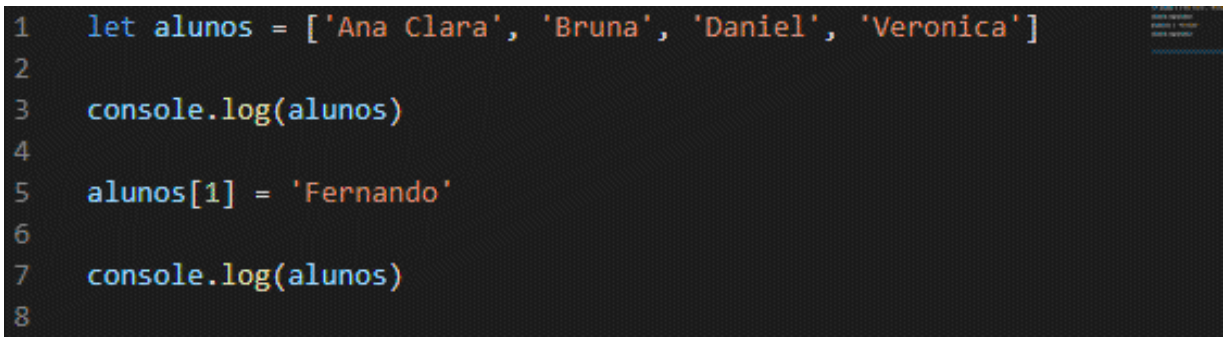
Nesse caso, referente a console.log( ) da linha 3 de nosso código será retornado Ana Clara, Bruna, Daniel, Veronica.

Da mesma forma, referente a console.log( ) da linha 5, será retornado apenas o elemento de posição de índice 3, nesse caso Veronica, haja visto que o índice de uma array sempre é iniciado em 0, logo, o quarto elemento da array é o de índice 3.

Manipulando elementos via índice

Entendido o conceito básico de que em uma array cada elemento de sua composição é mapeado e indexado para que assim possamos o manipular fazendo referência a sua posição de índice, é interessante lembrarmos que a partir do momento que temos a instância de um elemento da array, podemos aplicar sobre o mesmo todos os métodos que se aplicam a uma variável.

Outro ponto importante que devemos ter em mente é que, para realizar certas manipulações sobre elementos de arrays usando de operadores, os tipos de dados devem ser equivalentes para permitir tais operações.



Apenas para fins de exemplo, reaproveitando nossa array alunos, exibindo em tela seu conteúdo via console.log( ) como vemos na linha 3 do código o retorno será: Ana Clara, Bruna, Daniel, Veronica.

Na sequência, na linha 5 de nosso código, note que estamos fazendo referência apenas ao elemento de índice 1 da array atribuída a variável alunos, usando do operador de atribuição para inserir nesta posição de índice a string ‘Fernando’.

Exibindo em tela novamente o conteúdo de alunos via console.log( ), o retorno será: Ana Clara, Fernando, Daniel, Veronica, pois substituímos o elemento antigo de índice 1 por um novo elemento em forma de string.

O ponto a salientar é que, podemos manipular livremente os elementos de uma array conforme se faz necessário, porém, devemos nos atentar ao uso de operadores em tais elementos para evitar comportamentos indesejados. Supondo que quiséssemos inserir na posição de índice 1 o elemento ‘Fernando’ sem remover o elemento ‘Bruna’ que já estava em tal posição, deveríamos usar de uma função específica a este propósito e não um operador.

pop( ) / shift( )

Uma vez que temos uma array composta por uma série de elementos, podemos realizar algumas manipulações de forma simples e objetiva.

Para casos onde precisamos remover um elemento de nossa array podemos realizar tal ação através do método pop( ). Apenas lembrando que por parte da lógica deste tipo de dado, uma array segue por padrão o comportamento de uma fila LIFO (last in first out / último a entrar, primeiro a sair), onde quando aplicado o método pop( ), sempre o elemento a ser removido será o último elemento da array independentemente de seu tamanho.

```
1  let alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']
2
3  console.log(alunos)
4
5  alunos.pop()
6
7  console.log(alunos)
8
```

Seguindo com a mesma array do exemplo anterior, exibindo seu conteúdo via console.log( ) conforme definido na linha 3 do código, o retorno será Ana Clara, Bruna, Daniel, Veronica.

Em seguida aplicamos diretamente sobre a variável alunos o método pop( ), sem parâmetros mesmo.

Por fim, exibindo em tela novamente o conteúdo de alunos, agora nosso retorno será Ana Clara, Bruna, Daniel, em função do método pop( ) ter removido o último elemento da array, nesse caso Veronica.

```
1  let alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']
2
3  console.log(alunos)
4
5  alunos.shift()
6
7  console.log(alunos)
8
```

De forma alternativa ao método pop( ), fazendo uso do método shift( ) podemos remover o primeiro elemento / elemento de índice 0 de uma array.

Na linha 5 do código atualizamos a variável alunos aplicando o método shift( ) sobre a mesma.

Exibindo em tela o conteúdo de alunos através de console.log( ) o retorno será: Bruna, Daniel, Verônica. Lembrando que, uma vez que removemos o elemento de índice zero de nossa array, a array se auto-organizará, trazendo o elemento de índice 1 para a posição 0 e assim sucessivamente.

```
1  let alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']
2
3  console.log(alunos)
4
5  delete alunos[2]
6
7  console.log(alunos)
8
```

Apenas a nível de curiosidade, também alternativamente ao uso do método pop( ) é possível usar da palavra reservada ao sistema delete, que chama uma função interna que permite deletar qualquer

elemento de uma array independentemente de sua posição, porém com uma particularidade a ser considerada.

Diferentemente do esperado que seria, ao deletar um elemento de posição intermediária de uma array, os demais se rearranjarem, o que ocorre nesses casos é que na posição onde o elemento foi removido via delete ficará literalmente um espaço vazio.

Em nosso exemplo, na linha 5 de nosso código escrevemos uma sentença onde estamos a deletar o elemento de posição de índice 2 de nossa array alunos.

Exibindo em tela o conteúdo de alunos após essa modificação o retorno será: Ana Clara, Bruna, <1 empty item>, Veronica.

push( )

Exatamente o inverso da função pop( ) vista anteriormente, a função push( ) nos é útil para inserir um novo elemento em uma array.

Apenas salientando que enquanto pop( ) não recebia nenhum parâmetro por motivos óbvios, ao utilizarmos o método push( ) obrigatoriamente temos de parametrizar o mesmo com o devido elemento a ser inserido em nossa array.

```
1  let alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']
2
3  console.log(alunos)
4
5  alunos.push('Paulo')
6
7  console.log(alunos)
8
```

Seguindo com a mesma array, na linha 5 de nosso código aplicamos diretamente sobre a variável alunos o método push( ), parametrizando o mesmo com o nome Paulo em formato de string.

Exibindo em tela via console.log( ) o conteúdo de alunos, o retorno será: Ana Clara, Bruna, Daniel, Veronica, Paulo.

splice( )

Como vimos anteriormente, o método push( ) possui uma limitação básica onde simplesmente por meio do mesmo apenas conseguimos inserir novos elementos em nossa array seguindo a lógica de inserir elementos ao “final da fila”.

Contornando essa limitação temos o método splice( ), que nos permite a inserção (e remoção) de novos elementos em posições específicas intermediárias de índice, inclusive definindo via parâmetro se os elementos antigos serão removidos / substituídos ou simplesmente rearranjados, sendo assim um método muito mais útil para grande parte dos contextos de manipulação de arrays no que tange a inserção e remoção de elementos.

O método splice( ) por sua vez recebe três parâmetros em justaposição, sendo o primeiro a posição de índice para inserção, o segundo parâmetro referente ao número de elementos a serem removidos / substituídos, do terceiro parâmetro em diante sendo os próprios elementos a serem inseridos na array.

```
1   let alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']
2
3   console.log(alunos)
4
5   alunos.splice(1, 0, 'Tania')
6
7   console.log(alunos)
8
```

Nada melhor que a prática para que possamos entender tais conceitos.

Na linha 5 de nosso código instanciamos a variável alunos aplicando sobre a mesma o método splice( ), por sua vez parametrizado com

1, 0 ‘Tania’, ou seja, na posição de índice 1 será inserido a string ‘Tania’ sem substituir ou remover nenhum elemento.

Exibindo em tela o conteúdo de alunos via console.log( ) o retorno será: Ana Clara, Tania, Bruna, Daniel, Veronica, onde vemos que de fato Tania assumiu a posição 1 de índice, deslocando o antigo elemento Bruna para a posição 2 e assim sucessivamente.

```
let alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']

console.log(alunos)

alunos.splice(1, 1)

console.log(alunos)

```

Para usarmos do método splice( ) para remover certos elementos sem que suas posições de índice fiquem com elementos vazios, o que pode gerar comportamentos estranhos por parte da leitura da array, basta que usemos de seus parâmetros voltados a esse propósito.

No exemplo, usando a mesma array base dos exemplos passados, aplicando sobre a array alunos o método splice( ) parametrizado com 1, 1, basicamente estamos definindo que na posição de índice 1 será removido 1 elemento. Dessa forma, todos os demais elementos subsequentes se rearranjaram em suas devidas posições não ficando nenhum espaço vazio.

Exibindo em tela o conteúdo de alunos via console.log( ) o retorno nesse caso será: Ana Clara, Daniel, Veronica.

slice( )

Outro método de uso comum, mas não menos importante que os demais métodos abordados neste capítulo, é o método slice( ), o mesmo aplicável sobre strings também realiza fatiamentos em arrays nos mesmos moldes, bastando o parametrizar com um intervalo personalizado para que os elementos dentro desse intervalo sejam instanciados.

```
1   let alunos = ['Ana Clara', 'Bruna', 'Daniel', 'Veronica']
2
3   console.log(alunos)
4
5   alunos = alunos.slice(1, 3)
6
7   console.log(alunos)
8
```

Na linha 5 de nosso código, atualizando o conteúdo da variável alunos, usamos como atributo a própria variável aplicando sobre a mesma o método slice( ), nesse caso parametrizado com 1 e 3, irá retornar os elementos dentro desse intervalo de índice.

Exibindo em tela o conteúdo de alunos via console.log( ) o retorno será: Bruna, Daniel, pois o intervalo definido se inicia no elemento de índice 1, e termina no elemento anterior ao índice 3 da array.

sort( )

Para certos contextos podemos ter arrays de elementos aleatórios que precisam ser ordenados de alguma forma para cumprir algum pré-requisito. Existem diversas formas de se ordenar elementos de uma array, sendo a mais básica delas, aplicável tanto para arrays de elementos numéricos quanto de strings é o método sort( ).

```
let numeros = [99, 42, 10, 26, 27, 51, 89]

numeros = numeros.sort()

console.log(numeros)

```

Como exemplo agora temos uma variável de nome numeros que recebe em forma de array uma lista com números aleatórios.

Instanciando a variável numeros, atualizando a mesma ordenando seu conteúdo por meio do método sort( ), ao exibir o conteúdo da mesma via console.log( ) o resultado será uma nova array com os mesmos elementos, porém ordenados de forma crescente.

Nesse caso o retorno será: [10, 26, 27, 42, 51, 89, 99].

map( )

Por fim, um método embutido muito útil é o método map( ), pois através do mesmo é possível aplicar funções sobre cada elemento de uma array.

Em outras palavras, uma vez que temos uma array composta de elementos (independentemente de seu tipo), existe a possibilidade de se aplicar uma função sobre cada um dos elementos individualmente, gerando a partir dos mesmos uma nova array.

Raciocine que, por exemplo, temos uma array com alguns números aleatórios, e a partir dos mesmos queremos criar uma nova array, onde cada um dos números de origem tem seu valor elevado ao cubo. Ao invés de extrair elemento por elemento, aplicar a função e o inserir manualmente em uma nova array, podemos fazer isso diretamente sobre a array de origem através do método map( ).

```
function eleva_ao_cubo(valor, index, array) {
    return valor ** 3
}

let numeros = [99, 42, 10, 26, 27, 51, 89]

let numeros2 = numeros.map(eleva_ao_cubo)

console.log(numeros2)

```

Pondo em prática esse exemplo, logo de início criamos a função eleva_ao_cubo( ), que recebe por justaposição um dado/valor, seguido de um valor de índice e uma array de origem.

No corpo dessa função simplesmente criamos a sentença para que se retorno o valor repassado elevado a 3ª potência.

Em seguida temos uma variável de nome numeros, que é exatamente a mesma array do exemplo anterior, com seus respectivos valores numéricos.

Também é declarada uma nova variável de nome numeros2, que recebe como atributo para si o conteúdo de numeros após cada elemento seu ser mapeado e usado na função eleva_ao_cubo( ).

Exibindo em tela o conteúdo de numeros2 via console.log( ) temos uma nova array composta por: [970299, 74088, 1000, 17576, 19683, 132651, 704969].

filter( )

De aplicação semelhante a map( ), o método filter( ) normalmente é usado para criar uma nova array onde os elementos da array de origem são submetidos a alguma estrutura condicional ou de validação, de modo que os que atingirem os critérios estabelecidos serão filtrados e inseridos na nova array, enquanto os demais elementos são simplesmente desconsiderados.

```
function maior_que_50(valor, index, array) {
    return valor > 50
}

let numeros = [99, 42, 10, 26, 27, 51, 89]

let numeros2 = numeros.filter(maior_que_50)

console.log(numeros2)

```

Para nosso exemplo, reaproveitamos todo o código anterior, alterando apenas da função map( ) para a função filter( ), pois a nível de declaração e forma de parametrização ambas obedecem os mesmos critérios, além é claro de alterar nossa função personalizada, agora de nome maior_que_50( ), que como nome sugere filtrará os elementos os quais seu valor for maior que 50.

Exibindo em tela o conteúdo de numeros2 via console.log( ) é retornado uma nova array composta por: [99, 51, 89], únicos elementos que foram validados de acordo com a condição imposta.

--- // ---

Como mencionado anteriormente, map( ) e filter( ) são métodos muito parecidos porém com comportamentos distintos os quais devemos considerar dependendo do retorno que queremos para nossas funções.

```
function maior_que_50(valor, index, array) {
    return valor > 50
}

let numeros = [99, 42, 10, 26, 27, 51, 89]

let numeros2 = numeros.map(maior_que_50)

console.log(numeros2)

```

Apenas a nível de curiosidade, mantendo exatamente o mesmo código do exemplo anterior, alterando apenas filter( ) para map( ) e rodando o código, o retorno será: [true, false, false, false, false, true, true], pois no lugar de aplicar a função maior_que_50( ) validando e extraindo os elementos de valor numérico maior que 50 para uma nova array, por estarmos usando o método inadequado para esse propósito o que ocorre é apenas a verificação de cada elemento se o mesmo cumpre o critério estabelecido pela função, retornando um valor booleano true para os elementos que cumprirem o critério e false para os que não cumprirem o critério estabelecido.

reduce( )

Outro método embutido de funcionalidade interessante é o chamado método reduce( ), que por sua vez, de forma automatizada consegue retornar a soma dos elementos de uma array sem que para isso precisemos criar laços de repetição para percorrer elemento por elemento da array, mapeando o mesmo e somando e atribuindo a uma nova variável.

Através do método reduce( ) conseguimos, fornecendo os parâmetros certos, gerar um novo dado/valor referente a soma total dos elementos de uma array de origem.

```
1  function valor_total(total, valor, index, array) {
2      return total + valor
3  }
4
5  let numeros = [99, 42, 10, 26, 27, 51, 89]
6
7  let numeros2 = numeros.reduce(valor_total)
8
9  console.log(numeros2)
10
```

Como exemplo, criamos uma função de nome valor_total( ), que recebe em justaposição 4 parâmetros, sendo eles total, valor, index

e array, respectivamente.

Na sequência temos a variável numeros, mesma array dos exemplos anteriores.

Em seguida também temos a variável numeros2, porém dessa vez, a mesma recebe como atributo o resultado da função valor_total( ) aplicada via reduce( ).

Exibindo em tela via console.log( ) o conteúdo da variável numeros2, é retornado, como esperado, um valor referente a soma total dos elementos da array de origem, nesse caso, 344.

includes( )

Quando estamos criando manualmente nossas arrays, atribuímos como elementos para as mesmas dados / valores os quais temos ciência. Porém, em casos onde estamos a importar uma array de outras estruturas de código, pode ser que seu conteúdo esteja em algum formato implícito para o código atual.

Nesses casos, através do método includes( ) podemos facilmente verificar se um determinado dado / valor consta como elemento de uma array.

O método includes por sua vez necessita apenas de um parâmetro / argumento para realizar sua verificação, retornando um valor booleano caso tal dado tenha sido validado.

```
1   let numeros = [99, 42, 10, 26, 27, 51, 89]
2
3   console.log(numeros.includes(27))
4
```

Em nosso exemplo, diretamente como parâmetro para console.log( ) repassamos como parâmetro nossa variável numeros aplicando sobre a mesma o método includes( ), por sua vez parametrizado com o valor 27.

Nesse caso o retorno será true pois 27 é um dos elementos da array numeros.

PROGRAMAÇÃO ORIENTADA À OBJETOS

Conforme fomos vendo, capítulo após capítulo, a linguagem JavaScript é uma linguagem de programação muito fácil de se entender, uma vez que sua sintaxe é bastante simples e a partir dos recursos básicos que viemos aprendendo cada vez expandimos exponencialmente o leque de possibilidades de o que se fazer com tais estruturas de código.

Das estruturas de dados vistas até então, desde a simples declaração de variáveis até a criação e funções personalizadas, à medida que progredimos em nossos estudos entendemos meios e métodos de passar instruções a serem executadas pelo interpretador da linguagem, sem que necessariamente para isso o nível de complexidade aumentasse de acordo com a aplicação.

Fato é que JavaScript, assim como outras linguagens modernas (ex: Python) possui como característica a escrita de suas estruturas de dados voltadas a lógica, à sua parte funcional, diferentemente de outras linguagens onde grande parte do esforço se concentra em simplesmente desenvolver a escrita dos códigos, de modo que em JavaScript uma vez que dominamos o básico já conseguimos abstrair e resolver problemas computacionais de modo fácil e objetivo.

Dentro dessa linha de raciocínio, uma vez que entendemos o básico da chamada programação estruturada, podemos dar início ao entendimento da chamada programação orientada à objetos.

Quando falamos de programação orientada à objetos, independentemente da linguagem de programação a qual estamos usando, estamos falando de uma outra maneira de aplicar a lógica em nossos códigos. De forma alguma descartaremos o que foi aprendido até o momento, muito pelo contrário, assumindo uma outra abordagem para a escrita de nossos códigos estaremos

ampliando o horizonte de possibilidades, criando aplicações funcionais e eficientes.

Classes

Partindo para o paradigma da orientação à objetos aplicados a nossos códigos, um dos primeiros conceitos a serem entendidos é o de uma classe.

Raciocine que das estruturas base em JavaScript, uma das estruturas de dados mais dinâmicas e robustas que temos são as chamadas classes, objetos os quais usaremos para trabalharmos com certas abstrações em nossos códigos, de modo que uma classe será uma estrutura de dados com comportamento particular próprio assim como propriedade de poder ser usada como “molde” para criação de novas estruturas de dados a partir da mesma.

Uma classe por sua vez terá uma sintaxe própria assim como uma forma particular de guardar diferentes estruturas de dados em seu corpo de forma que cada uma dessas estruturas de dados será instanciável e possuirá suas próprias funcionalidades, podendo tais estruturas serem objetos de uso exclusivo da classe ou permitindo suas funcionalidades serem usadas / compartilhadas por outras estruturas de dados de fora da classe.

Tais conceitos podem parecer um tanto quanto confusos de início, porém com os exemplos certos poderemos entender todas as particularidades destes tipos de dados.

Antes mesmos de partirmos para o código, apenas para entendermos de fato o conceito apresentado nos parágrafos anteriores, vamos supor a criação de uma estrutura de dados que servirá para automatizar a criação de objetos referentes a informações de funcionários de uma empresa.

Nesta empresa fictícia, digamos que teremos diversos funcionários, cada um com uma especialização diferente, porém todos compartilhando de algumas características cadastrais como nome, sobrenome, data de admissão, cargo, salário e turno.

No formato convencional, usando de programação estruturada, estaríamos criando para esse contexto uma variável, atribuindo para a mesma algum tipo de contêiner de dados onde cada uma das informações referentes a cada funcionário estaria sendo inserida individualmente, repetindo esse processo manualmente para cada um dos colaboradores de tal empresa, o que acarretaria em um código no mínimo extenso, além de pouco funcional e eficiente.

Já fazendo uso de orientação a objetos, poderíamos criar uma classe com todos os devidos objetos referentes as características dos funcionários, de modo que para a criação de cada cadastro de cada funcionário, bastaria instanciar a classe alimentando a mesma com os dados do funcionário.

Dessa forma, além de um código reduzido, internamente a forma como o núcleo da linguagem irá trabalhar com cada atributo de cada objeto / variável se dará de modo muito mais eficiente.

```
let colaborador1 = {
    Nome: 'Fernando',
    Sobrenome: 'Feltrin',
    Admissão: '02/2005',
    Cargo: 'Eng. da Computação',
    Salário: 'R$3.546,68',
    Turno: 'Manhã + Remoto'
}

let colaborador2 = {
    Nome: 'Maria',
    Sobrenome: 'Brum',
    Admissão: '09/2012',
    Cargo: 'Desenvolvedora Senior',
    Salário: 'R$2.698,00 + Participação nos lucros',
    Turno: 'Integral'
}

console.log(colaborador1.Nome)
console.log(colaborador1.Cargo)

console.log(colaborador2.Nome)
console.log(colaborador2.Cargo)

```

Modo convencional, estruturado.

```
class Colaborador {
    constructor(nome, sobrenome, admissao, cargo, salario, turno) {
        this.nome = nome
        this.sobrenome = sobrenome
        this.admissao = admissao
        this.cargo = cargo
        this.salario = salario
        this.turno = turno
    }
}

```

Partindo para o modo orientado à objetos, visando criar uma estrutura que sirva como molde para a criação dos objetos

referentes a cada colaborador, vamos criar uma classe contendo a estrutura suficiente para tais objetos, e isto envolve algumas particularidades, um pouco chatas a primeira vista, confesso, porém que farão total sentido conforme entendemos suas aplicações.

Inicialmente, de acordo com a sintaxe, é criada a classe por meio da palavra reservada ao sistema class, seguido do nome personalizado da classe que estamos a criar, nesse caso, Colaborador.

Dentro do corpo dessa classe, repare que temos uma nova palavra reservada ao sistema, constructor, que é o identificador para o método construtor da classe, método este que como próprio nome sugere, serve para construir / criar objetos a partir dessa classe.

O método constructor por sua vez possui um campo delimitado entre parênteses, similar ao campo de variáveis usadas como parâmetros de uma função, onde estaremos inserindo os atributos de classe, referentes aos objetos que serão criados e alocados em memória toda vez que a classe for instanciada.

Nesse caso, inserimos os marcadores para nome, sobrenome, admissão, cargo, salario e turno, respectivamente e em justaposição, pois assim como os parâmetros de uma função, a ordem da declaração dos atributos de classe é levada em consideração pelo interpretador.

Na sequência, indentado ao método construtor constructor( ) temos o marcador this. seguido do nome do atributo de classe, recebendo um objeto homônimo como atributo.

Raciocine que this é um marcador / identificador que serve basicamente para que uma determinada variável / objeto / atributo de classe seja visível para todos os demais objetos e funções do código, algo relativo ao escopo do código que neste momento não precisamos entrar em detalhes.

Para cada atributo de classe é criado seu referente objeto com o marcador this. Dessa forma, criamos uma estrutura de dados a qual estaremos usando como esqueleto para a criação de novas

estruturas de dados de forma mais dinâmica, e this tornará esses objetos visíveis e iteráveis por parte de outras estruturas de código.

Novamente, apenas salientando que tal sintaxe, embora inicialmente pareça confusa ou até mesmo desnecessária, tratando-se da forma como iremos construir objetos para nossas aplicações é a forma mais otimizada, cabendo ao desenvolvedor apenas se adaptar ao uso dessa sintaxe.

Revisando, criamos a classe Colaborador através da palavra reservada class, dentro do corpo dessa classe usamos de um método de classe construtor constructor( ) criando a partir do mesmo as instâncias para os objetos os quais faremos uso.

```
12  let colaborador_1 = new Colaborador('Fernando',
13                                       'Feltrin',
14                                       '02/2005',
15                                       'Eng. da Computação',
16                                       'R$3.546,68',
17                                       'Manhã + Remoto')
18
19  let colaborador_2 = new Colaborador('Maria',
20                                       'Brum',
21                                       '09/2012',
22                                       'Desenvolvedora Senior',
23                                       'R$2.698,00 + Participação nos lucros',
24                                       'Integral')
25
26  console.log(colaborador_1.nome)
27  console.log(colaborador_1.cargo)
28
29  console.log(colaborador_2.nome)
30  console.log(colaborador_2.cargo)
31
```

De volta ao corpo / escopo geral de nosso código, declaramos uma variável de nome colaborador_1 que instancia a classe Colaborador, criando seus objetos a partir do esqueleto da classe.

Por parte de sintaxe, note que como atributo, a variável colaborador_1 recebe a palavra reservada new seguido do nome da classe, repassando como argumentos para a mesma, em

justaposição, alguns dados que alimentarão os atributos de classe nome, sobrenome, admissão, cargo, salario e turno, respectivamente.

A partir deste momento, a variável colaborador_1 possui como atributos os objetos nome:'Fernando', sobrenome:'Feltrin', admissão:'02/2005', cargo:'Eng. Da Computação', salario:'R$3.546,68' e turno:'Manhã + Remoto', de modo que podemos iterar sobre qualquer um destes objetos conforme necessidade.

Por exemplo, exibindo em tela via console.log( ) o conteúdo de colaborador_1.nome, nos será retornado Fernando, assim como usando como parâmetro colaborador_1.cargo será exibido em tela Eng. da Computação.

Para estes tipos de objetos básicos, como podemos perceber, a notação usada para iterar sobre os mesmos é muito parecida com a de um object, onde quando instanciamos uma determinada chave nos é retornado seu valor.

Se tratando especificamente de atributos de uma classe, instanciando seu objeto será retornado seu respectivo dado / valor atribuído.

```
1  class Colaborador {
2      nome_completo = nome_completo
3      escolaridade = escolaridade
4  }
5
6  let colaborador_1 = new Colaborador(nome_completo = 'Fernando Feltrin', escolaridade = 'Pós')
7
8  console.log(colaborador_1.nome_completo)
9
```

Se tratando de uma classe em sua forma mais básica, a mesma pode ou não conter um método construtor, de modo que podemos criar objetos literais dentro da classe os quais podem perfeitamente serem instanciados conforme a necessidade.

Apenas para fins de exemplo, criada uma classe de nome Colaborador que por sua vez possui em seu corpo apenas dois

objetos / atributos de classe, podemos fazer uso dos mesmos usando da notação de argumentos nomeados.

No escopo global do código é declarada uma variável de nome colaborador_1 que por sua vez instancia a classe Colaborador repassando como argumentos para a mesma alguns dados em forma de string para seus atributos de classe nomeados nome_completo e escolaridade, respectivamente.

Exibindo em tela através da função console.log( ) o conteúdo de colaborador_1.nome_completo, será retornado Fernando Feltrin.

Métodos de classe

Entendida a estrutura básica de uma classe, onde vimos que a mesma pode ou não ter uma método construtor de objetos, podemos avançar um pouco criando nossos próprios métodos de classe, que nada mais são do que funções criadas dentro do esqueleto de uma classe, também no intuito de oferecer uma certa funcionalidade tanto para os objetos internos à classe quanto para objetos do escopo global do código que usam de instâncias da classe.

Na prática, embora um método de classe seja uma função, a sintaxe entre ambas as formas são diferentes, assim como a maneira como repassamos argumentos / parâmetros para essa função / método de classe.

```
1   class Classe1 {
2       constructor(nome) {
3           this.nome = nome
4       }
5       soma(num1, num2) {
6           this.num1 = num1
7           this.num2 = num2
8           return num1 + num2
9       }
10  }
11
12  let variavel1 = new Classe1(nome = 'Fernando Feltrin')
13
14  console.log(variavel1.nome)
15
16  console.log(variavel1.soma(11, 43))
17
```

Diretamente ao exemplo, inicialmente criada a classe Classe1, dentro do corpo da mesma podemos ver que temos o método construtor constructor( ) que receberá obrigatoriamente um nome quando a classe for instanciada.

Em seguida temos um método de classe personalizado de nome soma( ), que por sua vez receberá dois argumentos num1 e num2, respectivamente, criando tais objetos para a classe e retornando a soma dos mesmos.

Fora da classe declaramos uma variável de nome variavel1 que instancia e inicializa a classe Classe1, passando como argumento para o atributo de classe nomeado nome uma string.

Note que o único campo obrigatório de preenchimento, no processo de se instanciar e fazer uso da classe, é o atributo de classe referente ao método construtor, enquanto o uso do método de classe soma( ) é totalmente opcional.

Através de console.log( ), exibindo em tela o conteúdo de variavel1.nome, será retornado Fernando Feltrin.

Da mesma forma, diretamente como parâmetro para console.log( ), repassando variavel1.soma( ) por sua vez parametrizada com 11 e 43, o retorno será 54.

O ponto a destacar aqui é que, uma vez que uma variável instancia uma classe, ela adquire todas as estruturas internas da mesma, mesmo que não faça uso de tais estruturas.

Em nosso exemplo, poderíamos muito bem simplesmente usar da classe para criar o objeto nome, ignorando o uso do método de classe soma( ) caso essa fosse a proposta.

```
class Classe1 {
    acao1() {
        console.log('Ação 1 sendo executada')
    }
    acao2() {
        console.log('Ação 2 sendo executada')
    }
}

let gatilho = new Classe1()

console.log(gatilho.acao1())

console.log(gatilho.acao2())

```

Dentro da possibilidades que a linguagem JavaScript nos oferece, no que diz respeito ao uso de orientação à objetos, é perfeitamente possível criar dentro do corpo de uma classe métodos de classe que não recebem nenhum argumento assim como não retornam nada para algum objeto.

Como demonstrado no exemplo, dentro do corpo de Classe1 temos dois métodos de classe acao1( ) e acao2( ), respectivamente, onde nem um nem outro recebe argumentos, porém quando instanciados, executam seu bloco de código, nesse caso, exibindo em tela uma mensagem ao usuário.

Através da variável gatilho instanciamos a classe Classe1, não repassando nenhum argumento / parâmetro pois a mesma não possui exigência nem por parte de método construtor nem de método personalizado de inserção de dados.

Via console.log( ), exibindo em tela o conteúdo de gatilho.acao1( ), é retornado: Ação 1 sendo executada. Exatamente o mesmo ocorre para gatilho.acao2( ) repassado como parâmetro para console.log( ).

```
class Login {
    constructor(nome = 'Usuário', id = '00000') {
        this.nome = nome
        this.id = id
    }
}

let funcionario1 = new Login()

console.log(funcionario1.nome)

let funcionario2 = new Login(nome = 'Fernando')

console.log(funcionario2.nome)

```

Muito semelhante ao que vimos criando parâmetros com valor padrão para nossas funções, se tratando dos marcadores para nossos atributos de classe as mesmas regras se aplicam, podendo assim inserir um valor padrão que é usado quando o usuário instancia a classe mas não fornece tal argumento.

Em nosso exemplo, criada uma classe de nome Login, dentro do corpo da mesma temos o método construtor e dentro do campo se inserção de argumentos para o mesmo temos nome e id, ambos com um valor padrão definido.

Fora da classe, é declarada uma variável de nome funcionario1, que instancia a classe Login, nesse caso, não repassando nenhum argumento para a mesma.

Exibindo em tela via console.log( ) o conteúdo de funcionario1.nome podemos ver que de fato foi retornado o valor padrão ‘Usuário’.

De modo parecido é criada uma nova variável de nome funcionario2, que agora instancia a classe Login, repassando como argumento para o atributo de classe nome a string ‘Fernando’.

Nesse caso, exibindo em tela o conteúdo de funcionario2.nome é retornado Fernando pois o valor repassado para o atributo de classe nome substituiu o valor padrão Usuário por Fernando.

Herança

Das particularidades relacionadas a programação orientada à objetos, o fato de uma classe servir como estrutura molde para criação de outras estruturas de dados nos permite uma série de recursos funcionais que na prática otimizam processos, consequentemente otimizando a performance de execução de nossos códigos.

Uma importante característica destas estruturas de dados é a de oferecer meios de iteração sobre seus objetos internos por qualquer outro objeto ou função de nosso código, e dentro dessa lógica, uma possibilidade muito útil é a de uma classe poder herdar outra, “importando” para si toda a estrutura de objetos de uma classe mãe para que não seja necessário criar novamente tais objetos a cada instância.

Na prática, uma vez que temos alguma classe com seus respectivos atributos e métodos de classe, os mesmos podem ser compartilhados à vontade para outras classes de modo que as classes herdeiras se tornem extremamente robustas sem que para isso haja a necessidade de se ter um código extenso e repetido. Uma vez que temos certas estruturas de código em formato orientado à objetos, as mesmas podem irrestritamente serem reutilizadas, desde que para um propósito coerente.

```
1   class Carros {
2       constructor(marca, modelo) {
3           this.marca = marca
4           this.modelo = modelo
5       }
6   }
7
8   class Carro1 extends Carros {
9       valor(valor) {
10          this.valor = valor
11      }
12      descricao() {
13          console.log(`Veículo Marca: ${marca}`)
14          console.log(`Veículo Modelo: ${modelo}`)
15          console.log(`Valor: ${valor}`)
16      }
17  }
18
19  let corsa1 = new Carro1(marca = 'Chevrolet', modelo = 'Corsa', valor = 'R$17.000')
20
21  console.log(corsa1.descricao())
22
```

Nada melhor que pormos em prática tais conceitos para que de fato possamos entender a lógica de herança em programação orientada à objetos.

Em nosso exemplo, logo na primeira linha é criada uma classe de nome Carros, dentro do corpo da mesma é criado seu método construtor, que nesse caso está a criar atributos de classe voltados a marca e modelo de um veículo.

Através de this são criados os devidos objetos, e assim temos nossa classe Carros pronta.

Dando continuidade a nosso código, na linha 8 é criada uma nova classe de nome Carro1, que por sua vez herda a classe Carros através do marcador extends.

Dentro do corpo da classe Carro1 temos um método de classe de nome valor( ) que por sua vez receberá um valor quando instanciado e inicializado.

Ainda dentro da classe Carro1 é criado outro método de classe, dessa vez de nome descricao( ) que quando instanciado e

inicializado, exibe em tela via console.log( ) três mensagens onde em cada uma, através de uma template string com sua respectiva máscara de substituição, são inseridos na mensagem de retorno dados oriundos dos objetos marca, modelo e valor.

Observe que o método de classe descricao( ) interage com dados / valores atribuídos aos objetos / atributos de classe marca, modelo e valor, onde marca e modelo são objetos da classe Carro1 herdados da classe Carros.

Em outras palavras, uma vez que a classe Carro1 herdou a classe Carros, todas as estruturas de código de Carros passaram a integrar a classe Carro1, de modo que para o interpretador, tais objetos são lidos e processados diretamente a partir da classe Carro1.

Alguns autores costumam pontuar o sistema de herança de classes como hierárquica, apenas para fins de abstração, pois para facilitar o entendimento lógico do encadeamento de classes e seus respectivos objetos, podemos perfeitamente imaginar as classes de origem como classes mãe e suas respectivas classes filhas que herdam suas características.

Nesse caso, a classe Carro1 é uma classe filha de Carros, logo, Carro1 possui todas as características de sua classe mãe somadas as suas próprias características, que podem ser objetos ou funções dependendo do contexto da aplicação.

Em seguida, na linha 19 de nosso código é declarada uma variável de nome corsa1, que por sua vez instancia a classe Carro1, repassando como argumentos para a mesma, para os devidos atributos de classe nomeados, textos em formato de string para marca, modelo e valor.

Por fim, exibindo em tela via console.log( ) o conteúdo da variável corsa1, chamando / aplicando o método de classe descricao( ) sobre a mesma, o retorno será:

Veículo Marca: Chevrolet

Veículo Modelo: Corsa

Valor: R$17.000

Para evitar que os tópicos deste livro fiquem muito extensos, obviamente os exemplos que são apresentados são um tanto quanto reduzidos, porém, raciocine dentro do tópico em questão, se tratando de heranças, imagine múltiplos objetos que iriam construir características de veículos, tendo que repetir a escrita de todas as variáveis necessárias, seria algo funcional mas nada eficiente, tendo em vista criar estruturas reaproveitáveis, assim como de características compartilhadas, fazer o uso de orientação à objetos tende a contornar todos estes problemas.

Classes como Funções

Em outros momentos foram mencionadas algumas vezes que em JavaScript tudo é função, no âmbito de que o modo como a linguagem é interpretada, cada tipo de objeto criado em meio ao código deve possuir um propósito, realizar alguma ação.

Dentro dessa linha de raciocínio, mesmo trabalhando em estruturas orientadas à objetos, cada estrutura de dado dessa modalidade é internamente uma função, logo, podemos abstrair o esqueleto de uma classe em forma de função para que assim possamos usufruir de todas suas funcionalidades de forma reduzia.

```
1   class Pessoa {
2       constructor(nome, idade, peso) {
3           this.nome = nome
4           this.idade = idade
5           this.peso = peso
6       }
7   }
8
9   let cliente1 = new Pessoa('Fernando', '33', '130kg')
10
11  console.log(cliente1.nome)
12  console.log(cliente1.peso)
13
```

A nível de código, vamos reescrever a estrutura de uma classe para fins de testes, validando se realmente ambas as formas se mostram funcionais.

Sendo assim, em nosso código é criada a classe Pessoa, que por sua vez possui um método construtor, que recebe três argumentos como atributos de classe, criando tais objetos e os alocando em memória assim que a classe é instanciada e alimentada com as devidas informações pelo usuário.

Dando sequência, novamente no corpo geral do código é declarada uma variável de nome cliente1, que instancia a classe Pessoa

repassando para a mesma três argumentos justapostos em formato de string.

Exibindo em tela via console.log( ) o conteúdo de cliente1.nome, assim como de cliente1.peso, nos é retornado Fernando 130kg.

```
function Pessoa2(nome, idade, peso) {
    const dados = {}
    dados.nome = nome
    dados.idade = idade
    dados.peso = peso
    return dados
}

let cliente2 = Pessoa2('Tania', '61', '58kg')

console.log(cliente2.nome)
console.log(cliente2.peso)

```

Reescrevendo nossa classe Pessoa para uma função de nome Pessoa2, logo de início definimos que a mesma receberá como parâmetros dados para nome, idade e peso.

Dentro do corpo dessa função criamos uma constante de nome dados que é um object inicialmente vazio.

Na sequência criamos dentro desse objeto as instâncias para nome, idade e peso que receberão tais dados quando o usuário chamar tal função parametrizando a mesma.

De volta ao escopo geral do código, declaramos uma variável de nome cliente2 que instancia e inicializa a função Pessoa2( ), parametrizando a mesma com dados escritos como strings em justaposição.

Por fim, exibindo em tela via console.log( ) o conteúdo de cliente2.nome e de cliente2.idade, nos será retornado Tania 58kg.

TÓPICOS COMPLEMENTARES

Closures

Muito do que vimos até então, capítulo após capítulo, foi desenvolvido focado diretamente na funcionalidade de cada estrutura de dados.

Nesse sentido, podemos observar que em JavaScript, diferentemente de outras linguagens, possui um interpretador bastante flexível onde podemos concentrar nossos esforços na lógica estrutural de cada código, enquanto uma série de processos internos é realizado implicitamente, garantindo todas as compatibilidades, comunicações e interações para cada um dos elementos de nosso código.

Dessa forma, quando estamos aprendendo sobre a linguagem JavaScript é natural buscar entender os mecanismos implícitos da mesma.

Muitas das estruturas internas a linguagem, ou implícitas para o desenvolvedor, de fato estão em uma camada interna propositalmente, visando que certas estruturas não sejam modificadas a ponto de gerar erros ou comportamentos inesperados. Porém, algumas estruturas, alguns conceitos e alguns comportamentos específicos podem ser estudados, e de acordo com seu propósito, dominar tais elementos pode contribuir para a construção de códigos mais eficientes.

Um dos conceitos implícitos, de suma importância para certos comportamentos de nossos códigos é o de escopo, nomenclatura usada quando nos referimos ao modo como certas estruturas de código são “visíveis” para o interpretador assim como iteráveis pelo mesmo.

Raciocine que existem formas de codificarmos certas estruturas funcionais e/ou lógicas para que as mesmas trabalhem dentro de um bloco específico ou que suas funcionalidades possam ser usadas por qualquer outro elemento ao longo do código.

Em outras palavras, quando criamos um novo arquivo em JavaScript, fato é que esse arquivo, mesmo em branco, possui uma série de configurações internas já repassadas ao interpretador da linguagem.

Quando começamos a escrever sentenças e blocos de código, assumimos implicitamente que toda e qualquer estrutura de dados no corpo do código é visível, instanciável e iterável por qualquer outro elemento do corpo geral do código.

Porém, como veremos no exemplo a seguir, é possível “encapsular” certas estruturas de código de modo que trabalhem apenas em torno de sua estrutura própria, tendo seus dados / valores e recursos invisíveis para os demais elementos do código, o que para algumas circunstâncias pode ser bastante útil.

Ditar o comportamento de uma estrutura de código para que trabalhe dentro de um escopo restrito / local, em JavaScript é uma prática chamada closure.

```
1   const variavel1 = 'Fernando'
2
3   function login() {
4       function logado() {
5           return variavel1
6       }
7       return logado()
8   }
9
10  let acesso = login()
11
12  console.log(acesso)
13
```

Partindo para a prática, vamos buscar entender o escopo de certos elementos e a partir disto como os mesmos são visíveis e iteráveis.

Em nosso código, logo na primeira linha é declarada uma constante de nome variavel1, que por sua vez recebe como atributo um texto em formato de string.

Na sequência é criada uma função de nome login( ), sem parâmetros mesmo, onde dentro da mesma temos uma nova função de nome logado( ) que retorna o conteúdo atribuído a constante variavel1.

Repare que neste encadeamento de funções (que poderia ser qualquer outra estrutura de código), estamos instanciando uma variável / constante que está fora da própria função, situada no corpo geral de nosso código. A nível de escopo, a constante variavel1 é um objeto situado no escopo global do código, logo, os dados / valores atribuídos a variavel1 são visíveis e iteráveis por qualquer outra estrutura do código.

Revisando o aninhamento de funções realizado, note que temos inicialmente a função login( ), que quando chamada por qualquer variável, irá executar sua função “filha” logado( ), retornando um dado/valor para variavel1.

Dando sequência, de volta ao escopo global do código é declarada uma variável de nome acesso, que instancia e inicializa a função login( ).

Exibindo em tela o conteúdo da variável acesso via console.log( ), o retorno será Fernando, dado atribuído a variavel1.

```
1   const variavel1 = 'Fernando'
2
3   function login() {
4       const variavel1 = 'Usuario'
5       function logado() {
6           return variavel1
7       }
8       return logado()
9   }
10
11  let acesso = login()
12
13  console.log(acesso)
14
```

Realizando uma pequena alteração, trazendo a constante variavel1 para dentro do corpo da função, a mesma passará a se comportar como uma variável local. Em outras palavras, em nosso exemplo, como temos duas variáveis / constantes, uma no corpo geral do código e outra no corpo da função, quando a função em si buscar pelo conteúdo dessa variável, irá considerar em primeira instância a que está situada dentro do corpo da função.

Mantendo todo o restante do código exatamente igual, ao executar o código nos é exibido em tela Usuario, referente a variável / constante local, de dentro da própria função.

Para fins didáticos, outra forma de assimilarmos tal comportamento é simplesmente raciocinar que, variavel1 do corpo geral do código é uma constante, com seu respectivo atributo, sendo um objeto iterável por todo e qualquer elemento.

Já a constante variavel1 de dentro do corpo da função login( ) é uma nova instância da constante variavel1, onde a mesma tem seu dado/valor atribuído atualizado de ‘Fernando’ para ‘Usuario’, de modo que pela regra da leitura léxica, o último valor lido em uma variável / constante é o que vale.

```
1   function login() {
2       const variavel1 = 'Usuario'
3       function logado() {
4           return variavel1
5       }
6       return logado()
7   }
8
9   let outra_variavel = variavel1
10
11  console.log(outra_variavel)
12
```

Além disso, a nível de closure, uma vez que temos uma constante variavel1 (ou qualquer outra) dentro do corpo de uma função, a mesma só será visível pelos demais elementos internos a função. Se tentarmos realizar qualquer iteração do escopo global do código

com uma variável / constante de dentro de uma função, um erro será gerado.

Apenas para fins de exemplo, simulando o erro, ao executar o código acima é exibido via terminal uma mensagem dizendo que variavel1 não existe. O ponto é que ela existe, porém só para a função login( ), sendo invisível para qualquer outra estrutura de dados.

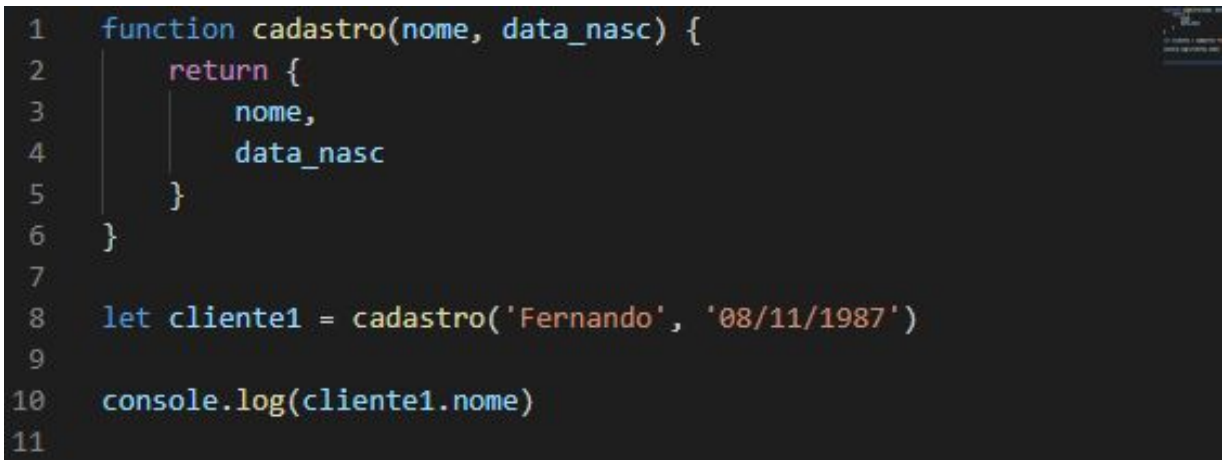
Função Factory

Quando estamos falando de funções em JavaScript, temos literalmente uma centena de possibilidades no que diz respeito tanto a forma de declaração / definição de uma função quanto ao comportamento esperado pela mesma.

O ponto é que quanto mais conhecemos tais possibilidades, mais otimizados tornamos nossos códigos fazendo uso de funções cada vez mais especializadas.

Nesse contexto, uma abordagem de funções não abordada no capítulo de funções, mas que guarda suas particularidades úteis para certas aplicações, é a criação das chamadas factory functions, funções simples, reduzidas e especializadas em criar novos objetos a partir de sua estrutura base.

Para situações onde temos de criar múltiplos abjetos, todos com algumas características em comum, podemos usar de uma função factory para criar o molde a ser replicado para todos os objetos. Na prática, o que difere esta estrutura base de função das demais é que a mesma diretamente retorna novos objetos com seus respectivos dados / valores para uma variável.



```
1   function cadastro(nome, data_nasc) {
2       return {
3           nome,
4           data_nasc
5       }
6   }
7
8   let cliente1 = cadastro('Fernando', '08/11/1987')
9
10  console.log(cliente1.nome)
11
```

Como exemplo, é criada de início uma função de nome cadastro( ) que por sua vez receberá obrigatoriamente e em justaposição dois parâmetros nome e data_nasc.

Dentro do corpo da mesma, sem nenhuma estrutura de código dedicada a alguma funcionalidade, é diretamente inserido um marcador para retorno, seguido da notação de chaves para que assim se retorne um object.

Como visto no capítulo dedicado aos tipos de dados, um object é um contêiner de dados onde é possível armazenar múltiplas variáveis de todo e qualquer tipo com seus respectivos atributos.

De volta ao escopo global do código, é criada uma variável de nome cliente1 que chama a função cadastro( ) parametrizando a mesma com ‘Fernando’ e ‘08/11/1987’.

Exibindo em tela via console.log( ) o conteúdo de cliente1.nome, será retornado Fernando, assim como qualquer outro objeto gerado pela função, desde que se use da notação de iteração para object, onde referenciamos uma determinada chave para obter seu respectivo valor.

Operador de desestruturação

Um operador mais avançado (e em função disso o mesmo não estar nos tópicos introdutórios sobre operadores) é o chamado destructuring.

Até mesmo por parte de sintaxe esse “operador” é escrito mais como um campo para parâmetros serão extraídos de um objeto do que um símbolo específico a ser identificado pelo interpretador.

O ponto é que, funcionalmente, um operador de desestruturação é um mecanismo do qual podemos extrair múltiplos elementos de um object sem que para isso seja necessário criar uma extensa estrutura de variáveis, operadores, condicionais e funções.

Como mencionado diversas vezes ao longo deste pequeno livro, uma vez que tudo em JavaScript é função, podemos tirar proveito dessa característica para produzir dados a partir de dados existentes, e esse processo pode ser reduzido fazendo o uso correto de desestruturações.

```
const funcionario1 = {
    nome: 'Fernando',
    idade: 34,
    funcao: 'Eng. da Computação',
    turno: 'Diurno',
    dados: {
        salario: 4932.55,
        admissao: '06/2019',
        auxilio: 'não',
        vale_transporte: 'não'
    }
}

const {nome: nome01, funcao: funcao01} = funcionario1

console.log(nome01)
console.log(funcao01)

```

Em nosso exemplo, declaramos uma constante de nome funcionario1, que por sua vez recebe uma série de dados em formato de object. Lembrando que um object em JavaScript, equivalente a um dicionário em Python e outras linguagens, é composto por dados dispostos em chaves:valores, sendo que os valores atribuídos as chaves podem ser qualquer tipo de dados (inclusive outro object). Sendo assim, compondo funcionario1 temos uma série de dados organizados como características de um funcionário fictício.

Na sequência, referente a linha 14 de nosso código temos o mecanismo de desestruturação, onde declaramos uma constante, diretamente seguido de um object, onde dentro do mesmo são criadas as instâncias de duas chaves nome e funcao que receberão dados dessas chaves homônimas, atribuindo seus respectivos dados / valores para as variáveis nome01 e funcao01, respectivamente. Finalizando a desestruturação temos o operador de atribuição seguido de funcionario1, que será a origem dos dados.

Em outras palavras, observe que o que está ocorrendo é que temos um object que receberá dados de funcionario1, dentro desse object instanciamos as chaves de funcionario1 as quais queremos extrair os dados, repassando os mesmos como atributos para variáveis que estão sendo criadas dentro do próprio object. Dessa forma temos, embora um tanto quanto abstrato, um mecanismo reduzido de extração de dados e geração de variáveis a partir de um object já existente em nosso código.

A partir do momento que o mecanismo de desestruturação é processado e concluído, as variáveis geradas são perfeitamente usáveis para outros fins. Nesse caso, exibindo em tela o conteúdo de nome01 via console.log( ) o retorno será Fernando, da mesma forma exibindo em tela o conteúdo de funcao01 o retorno será Eng. da Computação.

```
1   const funcionario1 = {
2       nome: 'Fernando',
3       idade: 34,
4       funcao: 'Eng. da Computação',
5       turno: 'Diurno',
6       dados: {
7           salario: 4932.55,
8           admissao: '06/2019',
9           auxilio: 'não',
10          vale_transporte: 'não'
11      }
12  }
13
14  const {nome: nome01, dados: {admissao: data_admissao01}} = funcionario1
15
16  console.log(nome01)
17  console.log(data_admissao01)
18
```

Como você deve ter reparado, como valor da chave dados de nosso object temos um nojo object com alguns campos organizados em chave:valor como a regra manda.

Através do mecanismo de desestruturação, para acessarmos essa camada de dados dentro do object original, basta que usemos da notação de object para ter acesso a tais dados.

Repare que, na linha 14 de nosso código, além da extração de dados realizada de nome e atribuída para nome01 como feito no exemplo anterior, agora estamos a entrar dentro do object associado a chave dados, mais especificamente extraindo dados da chave admissão, atribuindo tais dados a uma nova variável de nome data_admissao01.

Exibindo em tela via console.log( ) o conteúdo de nome01 será retornado Fernando, assim como exibindo em tela o conteúdo de data_admissao01 nos será retornado 06/2019.

MODULARIZAÇÃO

Quando estamos desenvolvendo nossas aplicações, é natural que certas estruturas de código acabem sendo alocadas para arquivos independentes, pois dessa forma tais estruturas de código podem ser acessadas, utilizadas e reutilizadas por diversos outros arquivos quando necessário, assim como tal prática facilita muito a manutenção destes blocos de código, uma vez que podemos realizar alterações nos mesmos sem que para isso tenhamos de alterar nossa estrutura de código principal.

Dentro dessa mesma linha de raciocínio uma prática muito comum é importar módulos de terceiros, que implementam funcionalidades que o JavaScript não possui ou não carrega por padrão quando iniciado.

Por parte sintática é necessário incluir alguns marcadores / palavras reservadas ao sistema para que ocorra a correta comunicação entre nosso arquivo de código principal e os arquivos referentes a módulos, permitindo o uso de blocos de código carregados a partir dos mesmos.

Contextualizando, imagine uma classe ou função criada de modo a ser utilizada por certas estruturas quando necessário. Supondo que tal estrutura de dados estivesse declarada ao longo do corpo de nosso código geral e não fosse utilizada em nenhum momento, ainda assim por parte de leitura léxica teríamos tais estruturas carregadas e pré-alocadas, afetando a performance de execução de nosso programa. Já se tais estruturas estivessem modularizadas em um arquivo independente, apenas quando se fizesse necessário tal arquivo seria importado, carregando e disponibilizando seu conteúdo.

```
class Pessoa {
    constructor(nome, sobrenome) {
        this.nome = nome
        this.sobrenome = sobrenome
    }
}

module.exports = Pessoa

```

Para fins de exemplo, criamos um arquivo de nome pessoa.js que será nosso módulo, dentro desse arquivo criamos uma classe de nome Pessoa que possui um método construtor, criando os objetos / atributos de classe nome e sobrenome, respectivamente.

Na sequência é criada a sentença module.exports = Pessoa, estrutura que fará com que a classe Pessoa não somente seja visível mas instanciável por outro arquivo que a importe. *Caso em nosso módulo houvessem outras estruturas a serem compartilhadas, como funções por exemplo, para cada uma dessas estruturas de código deverá ser criado o marcador module.exports.

```
const modulo_pessoa = require('./pessoa.js')

const pessoa1 = new modulo_pessoa('Fernando', 'Feltrin')

console.log(pessoa1.sobrenome)

```

Dando sequência é criado um novo arquivo de nome main.js, que será nosso arquivo principal da aplicação.

Logo na primeira linha, por convenção, são realizadas as devidas importações das bibliotecas, módulos e pacotes os quais faremos uso ao longo de nosso código. Nesse caso, declarada uma constante de nome modulo_pessoa, a mesma través do método require( ) carrega o arquivo pessoa.js (todo seu conteúdo).

Uma vez carregado o módulo pessoa, já podemos fazer uso de suas estruturas de código, sendo assim, declaramos uma nova constante, dessa vez de nome pessoa1, que usando de atributo o módulo

modulo_pessoa está usando da classe Pessoa interna ao mesmo, logo, no campo referente aos argumentos / atributos de classe são repassadas duas strings em justaposição.

Por fim, exibindo em tela o conteúdo de pessoa1.sobrenome via console.log( ) nos é retornado Feltrin.

PRÓXIMOS PASSOS

Em virtude de você ter chegado a este capítulo, podemos presumir que você tenha adquirido uma bagagem de conhecimento suficiente para que possa partir para uma ou mais de uma das inúmeras possíveis especializações dentro da linguagem JavaScript.

De fato, todo o conhecimento acumulado até o momento será sempre usado independentemente da ferramenta a qual você irá se especializar, haja visto que a lógica computacional e todas as estruturas de dados aprendidas nos capítulos deste livro sempre se aplicarão independentemente da aplicação.

Sendo assim, recomendo fortemente que você busque conhecer as ferramentas mais populares que implementam novas funcionalidades a linguagem JavaScript, seja para front-end ou back-end conforme sua preferência, de modo que aprenda os meios, métodos e particularidades de tais ferramentas.

Alguns exemplos de bibliotecas e ferramentas populares:

- ReactJS: Biblioteca voltada ao desenvolvimento simplificado de interfaces com o usuário.

- React Native: Framework dedicado ao desenvolvimento mobile.

- Express: Framework minimalista com ferramentas para construção de sites baseados em Node.

- AngularJS: Framework que oferece ferramentas para criação de aplicações web que rodam no cliente, incluindo interfaces e bancos de dados.

- Electron: Ferramenta para criação de aplicações multiplataforma baseadas em browser.

- jQuery: Ferramenta para desenvolvimento de aplicações baseadas em cliente.

- Bootstrap: Framework dedicado ao desenvolvimento front-end responsivo baseado em templates.

- Vue.js: Framework com propósito de facilitar o desenvolvimento de interfaces com o usuário.

CONSIDERAÇÕES FINAIS

Como costumo dizer ao final de meus livros, tudo o que tem um começo tem um fim, e tratando-se deste pequeno compêndio, avançamos alguns poucos passos na vasta gama de possibilidades que temos ao programar em JavaScript.

Espero que de fato a sua leitura tenha sido tão prazerosa quanto foi para mim escrever este humilde livro. Mais importante que isso, espero que tenha de fato contribuído com seus estudos e que tenha aprendido coisas novas a partir deste material.

Jamais pare de estudar, seja um eterno estudante de programação, especialize-se, e certamente estará à frente dos demais que por seus motivos ficam no entorno do básico.

Nos vemos em algum outro livro, curso ou treinamento, até lá... Um forte abraço!

Fernando Feltrin