



Curso 201

LÓGICA DE PROGRAMAÇÃO COM JAVASCRIPT

Sumário

1	Introdução à lógica	5
1.1	O que é lógica?	5
1.2	Lógica no dia a dia	5
1.3	O que é um algoritmo?	6
1.4	Algoritmos do cotidiano	6
1.5	Desafios de Lógica	6
1.6	Lógica no dia a dia do profissional de TI	9
1.7	Lógica proposicional resumida	10
1.7.1	Negação	10
1.7.2	Conjunção	10
1.7.3	Disjunção	11
1.7.4	Disjunção exclusiva	11
1.8	Exercícios	11
2	Introdução à programação	13
2.1	O que é um programa?	13
2.2	Como os programas são interpretados?	14
2.3	Linguagem de máquina x Linguagem de programação	14
2.4	Estrutura de um programa	16
2.5	Sistema Web x Sistema Desktop	16
2.6	Estrutura de um Sistema Web	17
3	Comandos básicos de entrada/saída	18
3.1	Escrevendo dados na saída	18
3.2	Lendo dados da entrada	20
4	Variáveis	24
4.1	O que é uma variável?	24
4.2	Nomes válidos	24
4.3	Tipos primitivos	25
4.3.1	Alguns tipos nativos do JavaScript	25

4.4	Operadores aritméticos	26
4.5	Usando JavaScript para demonstrar na prática	26
4.6	Exercícios	29
5	Estrutura de decisão	31
5.1	Se / senão	31
5.2	Se encadeado	34
5.3	Operadores Lógicos	35
5.3.1	Operadores de comparação	36
5.3.2	Operador ternário de decisão	37
5.4	Exercícios	39
6	Estruturas de repetição	41
6.1	Comando while	41
6.1.1	do ... while	43
6.2	Comando for	44
6.3	Controlando o fluxo	46
6.4	Exercícios	48
7	Funções	49
7.1	Declaração / Chamada de uma função	49
7.2	Passagem de parâmetros	51
7.2.1	Funções sem nenhum parâmetro	51
7.2.2	Funções com um ou mais parâmetros	52
7.3	Retorno	53
7.4	Escopo da função	55
7.5	Exercícios	58
8	Vetores	60
8.1	Criação de Vetores	60
8.2	Manipulação de Vetores	61
8.2.1	Adicionando itens ao vetor	61
8.2.2	Removendo itens do vetor	63
8.2.3	Criando um novo vetor a partir de um existente	64
8.2.4	Filtrando elementos de um vetor	66
8.2.5	Outras operações com vetores	67
8.3	Revisitando repetições	68
8.4	Exercícios	69
9	Projeto: Casa de Câmbio	71
9.1	As moedas	71
9.2	A casa de câmbio	72

9.3	Moedas e seus valores	73
9.4	Implementação	73
10	Desafio: Jogo de Damas em JavaScript	76
10.1	Jogadores	76
10.2	Regras	77
10.3	Objetivos	78
10.4	Implementação	79

1

Introdução à lógica

O que é lógica?

Lógica é a forma de raciocínio que se propõe a determinar o que é verdadeiro ou falso de acordo com premissas previamente acordadas/estabelecidas. Estas premissas podem ser fatos ou argumentos supostos como verdadeiros.

O raciocínio lógico se dá por uma argumentação consistente que visa a harmonia de raciocínio bem como uma ordem sequencial ou alguma outra forma de conexão entre os fatos. As conexões entre os fatos e a harmonia de raciocínio geram coerência no discurso, que pode ser avaliado e aprovado por diferentes entidades de diferentes contextos.

Lógica no dia a dia

Todos nós utilizamos de algum pensamento lógico no nosso dia a dia. Por exemplo, se você trabalha e precisa começar o expediente em um certo horário, então você provavelmente acorda algumas horas antes do expediente para que tenha tempo para tomar o café da manhã, se arrumar, etc. Se o procedimento de se arrumar é o mesmo todos os dias, então você provavelmente acorda sempre no mesmo horário.

Precisar estar fisicamente no local de trabalho também interfere em suas decisões lógicas: se você trabalha no modelo *home-office*, você provavelmente acorda mais tarde do que quem precisa estar no local físico do trabalho.

Outro exemplo mais prático pode ser:

Se você se encontra em um lugar, como uma casa ou um escritório, onde há energia elétrica, você pode decidir se acende ou não as luzes de acordo com a luminosidade externa disponível. Faz sol e há janelas? Talvez não seja necessário acender as luzes. Está de noite, ou escuro por não haver janelas? Acender as luzes pode lhe ajudar a enxergar.

O que é um algoritmo?

Um algoritmo é uma sequência lógica e finita de instruções que são precisas e não-ambíguas. Algoritmos, além de precisarem ser eficientes, possuem um objetivo final, como, por exemplo, devolver a soma de 2 números. Algoritmos podem ser iterativos ou recursivos, seriais ou paralelos, e podem ir desde uma receita de sanduíche ou café até a ordenação de uma grande coleção.

Algoritmos do cotidiano

Um exemplo de algoritmo do cotidiano é o preparo de café utilizando uma máquina de café. Tendo em mente que as variáveis das quais precisamos saber para o nosso exemplo são da presença de água e pó de café na máquina, podemos escrever o nosso algoritmo da seguinte forma:

```
abra o compartimento de pó de café
se não houver pó de café
então:
    preencha o compartimento com pó de café
feche o compartimento de pó de café

abra o compartimento de água
se não houver água
então:
    preencha o compartimento com água
feche o compartimento de água

aperte o botão liga/desliga

até que o café esteja pronto:
    aguarde

sirva o café
```

Desafios de Lógica

Existem diversos jogos de lógica disponíveis gratuitamente na internet hoje em dia. Em especial, jogos do Brainzilla podem ser bons para praticar o raciocínio lógico. Nesta seção, resolveremos juntos um de equações lógicas.

1. Começamos com uma tabela 9x9 vazia

	1	2	3	4	5	6	7	8	9
A									
B									
C									
D									
E									
F									
G									
H									
I									

$$\begin{aligned} A + D + E &= 13 \\ E + G &= 7 \\ E + I &= 4 \\ A + H &= 10 \\ A + F + H &= 19 \\ A + C &= 13 \end{aligned}$$

- Temos as equações ao lado, que usaremos para descobrir os valores e preencher os buracos na tabela
 - Cada variável (de A a I) pode assumir apenas um valor de 1 à 9, sem repetições
2. Começamos descobrindo F ao isolá-lo da equação que o usa
1. Temos que $A + H = 10$ e $A + F + H = 19$
 2. Logo sabemos que $10 = 19 - F$, portanto $F = 9$

	1	2	3	4	5	6	7	8	9
A									×
B									×
C									×
D									×
E									×
F	×	×	×	×	×	×	×	×	✓
G									×
H									×
I									×

$$\begin{aligned} A + D + E &= 13 \\ E + G &= 7 \\ E + I &= 4 \\ A + H &= 10 \\ A + F + H &= 19 \\ A + C &= 13 \end{aligned}$$

- Isso também significa que nenhuma outra variável pode assumir o valor 9.
3. Podemos perceber também que há 4 equações semelhantes:
1. $A + H = 10$ e $A + C = 13$
 - Sabemos que nem A e nem H podem ser 1 ou 9, pois F já é 9

	1	2	3	4	5	6	7	8	9
A	×								×
B									×
C									×
D									×
E									×
F	×	×	×	×	×	×	×	×	✓
G									×
H	×								×
I									×

$$A + D + E = 13$$

$$E + G = 7$$

$$E + I = 4$$

$$A + H = 10$$

$$A + F + H = 19$$

$$A + C = 13$$

- Ao isolarmos H e C , podemos descobrir que C é maior que H por 3 unidades.
2. $E + G = 7$ e $E + I = 4$
- Ao isolarmos G e I , podemos descobrir que
 1. G é maior que I por 3 unidades
 2. E é 1 e I é 3
 - G é 6
 3. ou E é 3 e I é 1
 - G é 4
4. Neste passo, temos que

	1	2	3	4	5	6	7	8	9
A	×	×	×	×	×				×
B									×
C	×	×	×	×				×	×
D									×
E		×		×	×	×	×	×	×
F	×	×	×	×	×	×	×	×	✓
G	×	×	×		×		×	×	×
H	×				×	×	×	×	×
I		×		×	×	×	×	×	×

$$A + D + E = 13$$

$$E + G = 7$$

$$E + I = 4$$

$$A + H = 10$$

$$A + F + H = 19$$

$$A + C = 13$$

Porque:

- C não pode estar no intervalo $[1..4]$ pois é maior que H em 3 unidades e H não pode ser 1 (ou seja, H precisa começar em 2, no mínimo)
- H não pode estar no intervalo $[6..8]$ pois é menor que C em 3 unidades e C não pode

- ser 9 (ou seja, c pode ser no máximo 8)
- A e H não podem ambos assumir o valor 5
 - Logo, c não pode ser 8
 - A não pode estar no intervalo $[2..4]$ pois a soma com H precisa dar 10
5. Por fim, temos a equação $A + D + E = 13$
1. Se substituirmos E por 3, temos que $A + D = 10$
 - Não é possível, porque isso significaria que D e H assumem o mesmo valor
 2. Se substituirmos E por 1, temos que $A + D = 12$
 - Se E é 1, então I é 3 e G é 6
 - A não pode ser 6
 3. Se substituirmos A por 7, $D = 12 - 7 = 5$
 - Não é possível, porque $A + C = 13$, $C = 13 - 7 = 6$ e G é 6
 4. Se substituirmos A por 8:
 - $D = 12 - 8 = 4$
 - $H = 10 - 8 = 2$
 - $C = 13 - 8 = 5$
6. Por exclusão, B é 7

	1	2	3	4	5	6	7	8	9
A	×	×	×	×	×	×	×	✓	×
B	×	×	×	×	×	×	✓	×	×
C	×	×	×	×	✓	×	×	×	×
D	×	×	×	✓	×	×	×	×	×
E	✓	×	×	×	×	×	×	×	×
F	×	×	×	×	×	×	×	×	✓
G	×	×	×	×	×	✓	×	×	×
H	×	✓	×	×	×	×	×	×	×
I	×	×	✓	×	×	×	×	×	×

$A + D + E = 13$
 $E + G = 7$
 $E + I = 4$
 $A + H = 10$
 $A + F + H = 19$
 $A + C = 13$

Lógica no dia a dia do profissional de TI

Ter um certo domínio em raciocínio lógico é imprescindível para o profissional de TI. Diariamente, o profissional terá de não apenas usar deste raciocínio para codificar, como também para tomar as decisões que respondam à perguntas como, por exemplo, “como funciona”, ou “por que funciona desta forma”, entre outras.

A lógica também é importante ao profissional de TI quando é necessário decidir qual tecnologia

deve ser usada para um certo fim. Usar um banco de dados relacional ou não-relacional? Uma linguagem de programação compilada ou interpretada? Um sistema operacional de código aberto ou fechado? Ainda que muitas tecnologias possam ser usadas para atingir um mesmo objetivo, existem casos em que uma tecnologia T pode ser melhor do que uma tecnologia U porque T facilita um procedimento que seria inviável ou mais difícil de executar utilizando U .

Lógica proposicional resumida

A lógica proposicional é uma maneira de usar o raciocínio lógico para validar se certas proposições, ou declarações, possuem um valor verdadeiro ou falso. O estudo de lógica proposicional se estende muito mais do que o que veremos aqui, mas a princípio é suficiente para que estudemos e entendamos as quatro tabelas-verdade a seguir para este curso.

É comum que usemos letras como p , q , r e s para estas demonstrações. Daremos sentido à elas quando necessário.

Negação

A negação de um valor verdadeiro é o valor falso; a negação de um valor falso é o valor verdadeiro.

p	$\neg p$ (não p)
V	F
F	V

Ou seja, se p é **eu dirijo**, então $\neg p$ é **eu não dirijo**.

Conjunção

A conjunção de dois valores só é verdadeira quando ambos os valores são verdadeiros.

p	q	$p \ \&\& \ q$ (p e q)
V	V	V
V	F	F
F	V	F
F	F	F

Se p é **eu tenho 18 anos ou mais** e q é **eu possuo habilitação**, então podemos dizer que $p \ \&\& \ q$ é **eu posso dirigir**: só posso dirigir se tenho 18 anos ou mais e possuo habilitação.

* Não faz sentido ter menos de 18 anos e ser habilitado, mas ignoremos essa parte pelo bem do exemplo.

Disjunção

A disjunção de dois valores é verdadeira quando pelo menos um dos valores é verdadeiro.

p	q	$p \vee q$ (p ou q)
V	V	V
V	F	V
F	V	V
F	F	F

Se p é **sou graduado em letras** e q é **sou graduado em ciência da computação**, então $p \vee q$ é **sou graduado em letras e/ou ciência da computação**: é verdadeiro desde que eu seja graduado em pelo menos uma das duas.

Disjunção exclusiva

A disjunção exclusiva de dois valores só é verdadeira quando apenas um deles é verdadeiro.

p	q	$p \oplus q$ (ou p ou q)
V	V	F
V	F	V
F	V	V
F	F	F

Se p é **vou viajar durante as férias** e q é **vou ficar em casa durante as férias**, então $p \oplus q$ é **durante as férias ou vou viajar ou vou ficar em casa**: só é verdadeiro se apenas uma delas for verdadeira.

Exercícios

- Escreva dois algoritmos. Represente a sua rotina matinal e noturna em cada um. Tente ser preciso nos passos e não se prenda à uma "sintaxe". Pense na sua rotina como uma "receita" do que fazer nestes períodos. Exemplo:

```
tirar os sapatos
entrar em casa
andar até o banheiro
```

```

lavar as mãos na pia (importante durante a pandemia)
andar até a cozinha
pegar um copo
fazer duas vezes:
    até que o copo esteja três-quartos cheio:
        colocar água no copo
    até que o copo esteja vazio:
        beber água do copo
    aguardar dois segundos
lavar o copo
...

```

2. Complete as tabelas-verdade abaixo:

1.	p	q	r	!r	p & q & !r
	V	V	V		
	V	V	F		
	V	F	V		
	V	F	F		
	F	V	V		
	F	V	F		
	F	F	V		
	F	F	F		

2.	p	q	r	s	!q	!s	(p & !q) ^ (r !s)
	V	V	V	V			
	V	V	V	F			
	V	V	F	V			
	V	V	F	F			
	V	F	V	V			
	V	F	V	F			
	V	F	F	V			
	V	F	F	F			
	F	V	V	V			
	F	V	V	F			
	F	V	F	V			
	F	V	F	F			
	F	F	V	V			
	F	F	V	F			
	F	F	F	V			
	F	F	F	F			

2

Introdução à programação

O que é um programa?

Um programa é um conjunto de instruções escrito em uma linguagem que o computador entende. Essas instruções descrevem uma tarefa que deve ser realizada por ele.

Quando dizem *programa*, podem estar se referindo aos arquivos que contêm o código-fonte ou ao executável.

O código-fonte de um programa pode ser compilado ou interpretado.

Programas *compilados* são programas cujo código-fonte é compilado. A compilação de um programa envolve validar a sua sintaxe, garantir que todas as variáveis usadas existem, garantir que todas as chamadas de funções estejam corretas e, se suportado pelo compilador, verificar se pode haver possíveis erros de valores como, por exemplo, valores nulos que não poderiam ser nulos. O compilador então junta todas as peças necessárias e reescreve o programa em um formato binário que o computador (ou a máquina) entende. Este binário, comumente chamado de executável, pode ser redistribuído e executado em outras máquinas sem que precise passar pelo processo de compilação novamente.

Programas *interpretados*, por sua vez, são programas cujo código-fonte é lido e interpretado no momento de sua execução. Toda a validação que o compilador faz com linguagens compiladas é feita pelo que chamamos de interpretador para linguagens interpretadas, e porque é feita no momento de execução, é necessário que o computador (ou máquina) possua o programa interpretador para que o programa possa ser redistribuído.

Linguagens interpretadas geralmente têm execução mais lenta justamente por conta deste passo adicional na execução.*

* Hoje em dia, muitas linguagens interpretadas contam com um modo de compilação chamado *Just In Time*, ou Compilação na Hora. Esta é uma estratégia que compila o código-fonte para um código binário que pode ser reaproveitado pelo interpretador em execuções consecutivas. A compilação é refeita automaticamente caso o conteúdo do código-fonte seja alterado.

Como os programas são interpretados?

O sistema operacional, ao receber a instrução para executar um programa, começa carregando-o em memória. Ele organiza as partes do programa em segmentos ou seções e logo depois começa a executar as instruções do programa, uma após a outra. A execução do programa pode seguir até o final ou terminar precocemente se ocorrer algum erro de software ou hardware.

Linguagem de máquina x Linguagem de programação

A linguagem de máquina é uma linguagem composta por uma sequência de *bytes*, onde cada sequência pode significar uma instrução ou até mesmo um valor. Essas sequências têm tamanhos definidos e normalmente diferem de uma marca de processador à outra.

Linguagens de máquina são praticamente ilegíveis, visto que seria necessário ler o código junto a um manual com cada uma das instruções, assim como se perder nele seria fácil também. Existem também as linguagens de montagem, que oferecem uma abstração em cima das linguagens de máquina ao usar palavras em inglês que são traduzidas ("montadas") para as instruções binárias, uma por uma.

Já linguagens de programação possuem uma camada inteira de abstração em cima das linguagens de máquina que faz com que escrever um programa seja parecido com escrever uma redação em inglês: elas possuem uma sintaxe própria, expressões, cláusulas e instruções que podem ser muito mais facilmente lidas e entendidas do que linguagens de máquina ou de montagem. A primeira linguagem de programação que se tornou grande (e se tornou a gigante cujos ombros são usados até hoje para construir novas tecnologias) e substituiu as linguagens de montagem em grande escala é C.

Abaixo seguem exemplos de como somar dois números, 3 e 7, usando três variáveis, a, b e c, e exibir o resultado e se ele é maior que 10 ou menor ou igual, em uma pseudo linguagem de montagem, em C, e em Python:

Pseudo linguagem de montagem:

```
main:
mov a, 3
mov b, 7
mov c, a
add c, b
show c
cmp c, 10
jle let
gtt:
show "maior que 10"
jmp finish
let:
show "menor ou igual a 10"
finish:
ret 0
```

C:

```
#include <stdio.h>

int main(void) {
    int a = 3;
    int b = 7;
    int c = a + b;

    printf("%d\n", c);

    if (c > 10) {
        printf("maior que 10\n");
    }
    else {
        printf("menor ou igual a 10\n");
    }

    return 0;
}
```

Python:

```
a, b = 3, 7
c = a + b

print(c)

if c > 10:
    print("maior que 10")
else:
    print("menor ou igual a 10")
```

Estrutura de um programa

A estrutura de um programa pode variar de acordo com variáveis, como linguagem de programação e paradigma, mas em essência pode ser resumida em duas partes:

1. Execução
2. Dados

Os dados são os valores que o programa usa. Números inteiros ou de ponto flutuante, caracteres, booleanos (verdadeiro e falso), vetores e também as estruturas de dados que nós mesmos construímos para satisfazer as nossas necessidades. Algumas linguagens de programação já oferecem diversas dessas estruturas para a nossa conveniência.

Já a execução é todo código executável do nosso programa, e também o que usa os dados mencionados anteriormente. A grosso modo, são funções, ou blocos de código parametrizáveis que podem ser reusados de acordo com as necessidades do programador.

A parte executável é a que engloba expressões, comandos, estruturas de decisão e repetição e tudo mais. Em uma analogia, é como se programas fossem orquestras: os músicos (e os instrumentos) são os dados e o maestro o executor, orientando e usando cada músico e instrumento no momento e com a intensidade com que devem ser usados.

Sistema Web x Sistema Desktop

Olhando para o quadro geral, um sistema web e um sistema desktop não são muito diferentes. Ambos são constituídos por:

- Uma interface gráfica com botões, campos de texto e menus drop-down.
- Uma forma de armazenar e carregar dados.
- Um fluxo de negócio, que nos leva de uma tela à outra de forma consistente de acordo com o contexto.

Então pode parecer que ambos os tipos de sistemas funcionam da mesma forma e que, no final das contas, “tanto faz” qual escolhemos fazer ou usar. Mas sistemas web e desktop possuem algumas diferenças fundamentais que podem influenciar nessa decisão.

- Sistemas web só precisam ter os programas envolvidos em uma ou algumas máquina(s), enquanto sistemas desktop precisam ter os programas em todas as máquinas que o usarão.
Uma atualização de software de um sistema desktop envolve (re)instalar o software em todas as máquinas que precisam dele.
- Sistemas web são multi-plataforma por padrão, enquanto sistemas desktop normalmente

são desenvolvidos para uma plataforma específica. Para usar um sistema web, tudo o que você precisa é um navegador, como Mozilla Firefox ou Google Chrome, o endereço e, possivelmente, conexão à internet. Por outro lado, um sistema desktop muito provavelmente funciona apenas para a(s) plataforma(s) para a(s) qual(is) ele foi desenvolvido: Linux, Windows, MacOS ou outras.

- Sistemas desktop podem ser mais especializados do que sistemas web. Sistemas desktop podem ser usados para ler e modificar propriedades do hardware da máquina como, por exemplo, criar e recuperar backups. Algo que um sistema web generalizado não é capaz de fazer.

Estrutura de um Sistema Web

Um sistema web normalmente consiste em um número de tecnologias, algumas mais comuns que outras. Os primeiros três tipos de tecnologia listados a seguir são praticamente os pilares de sistemas web.

Em primeiro lugar temos o servidor web, que é o responsável por abrir as portas do sistema ao mundo. Ele é quem recebe os pedidos do navegador quando acessamos algum endereço e os redireciona aos processos responsáveis como, por exemplo, o servidor do app. O servidor web também é responsável por servir arquivos estáticos como arquivos CSS e JavaScript.

O servidor do app é uma das peças mais importantes do quebra-cabeças, pois é este que contém todo o código, todas as regras de negócio da nossa aplicação. As aplicações também renderizam arquivos HTML de uma forma relativamente dinâmica. Quando o arquivo HTML não precisa de dinamicidade, ele pode ser servido como um arquivo estático pelo servidor web.

A terceira peça essencial é o servidor de banco de dados, que providencia uma forma de armazenar e resgatar dados persistentes. Perfis de usuários em redes sociais, histórico de compras de e-commerces e listas de favoritos de serviços de streaming são todos armazenados em bancos de dados.

Adicionalmente, pode também haver serviços de caching, filas, pubsub, monitoramento, entre outros, que não necessariamente constituem a fundação de sistemas web, mas, ainda assim, ajudam e muito os desenvolvedores e especialistas de infraestrutura.

3

Comandos básicos de entrada/saída

Em programação, *entrada e saída* (*input and output*, ou simplesmente **IO**) é um dos tópicos mais importantes, pois, sem elas, nossos programas seriam estáticos e fariam sempre a mesma coisa sem nenhuma perspectiva de mudança. Sem entrada e saída não haveria comunicação com serviços externos, como bancos de dados e, mais importante, com os usuários destes programas.

Dizemos que entrada e saída são os responsáveis pela comunicação com o mundo externo.

Neste capítulo, trataremos de entrada e saída no que diz respeito à interação do usuário com o programa. Veremos como é possível o software usar dados providos pelo usuário - nós - e realizar cálculos ou tomar decisões com base neles.

Suponhamos que um programa calcule a idade que uma pessoa atinge ou atingirá no ano atual. Tal pessoa precisa entrar com o seu ano de nascimento para que o programa possa calcular a diferença entre o ano atual e seu ano de nascimento. Se o ano atual for 2022 e o ano de nascimento do usuário for 1992, o programa deve exibir 30 na saída.

Escrevendo dados na saída

Linguagens de programação diferentes oferecem formas diferentes de exibir dados na saída padrão do computador - normalmente, a saída padrão é a tela. Como todos os exemplos deste curso usam JavaScript, é com esta linguagem que mostraremos como exibir dados na tela.

JavaScript nos oferece uma função chamada `console.log`. Aprenderemos mais sobre funções nos próximos capítulos; por ora, basta saber que uma função é um “código pronto para ser usado”. Podemos passar qualquer dado para a função `console.log` e ela se encarregará de exibí-lo na tela. Abra o console do desenvolvedor no seu navegador e vamos testar alguns exemplos.

- Se você usa Google Chrome, aperte `Ctrl + Shift + I` e selecione a aba `Console`.
- Se você usa Mozilla Firefox, aperte `Ctrl + Shift + K`; a aba `Console` *deve* estar selecionada por padrão.

Digite:

```
console.log("Olá, mundo!");
```

Seguido de `Enter`. O seu navegador deve lhe mostrar algo parecido com a imagem a seguir:

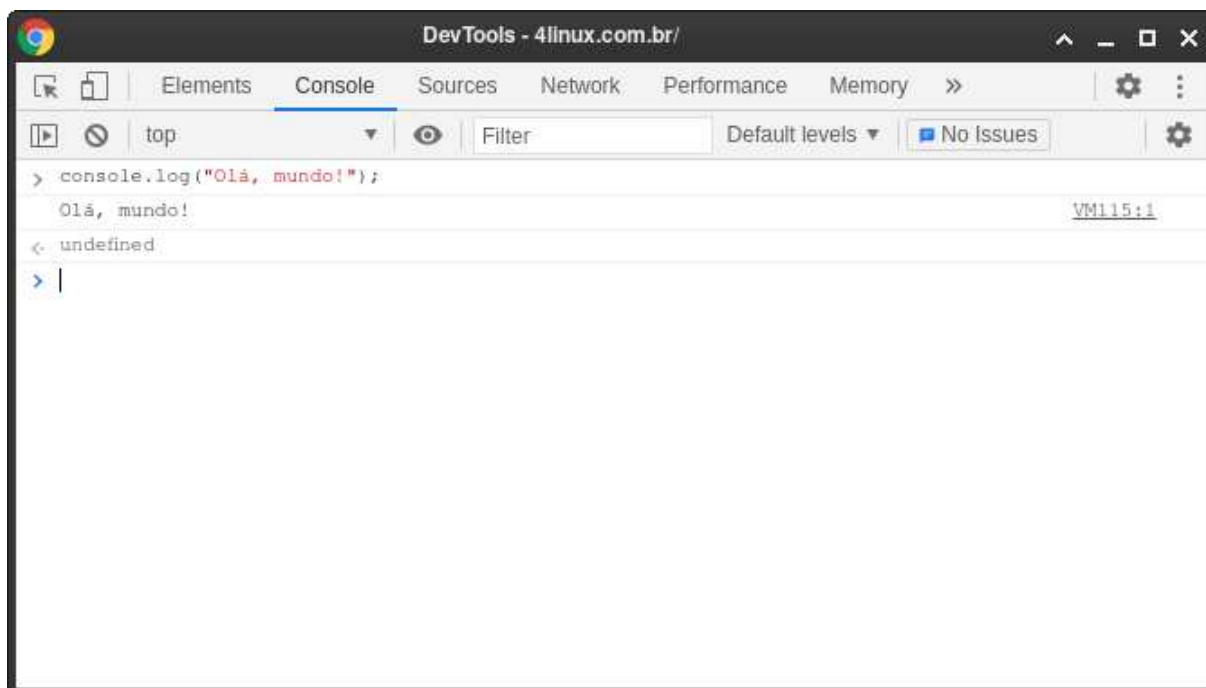


Fig. 3.1: console olá mundo

Alternativamente, o JavaScript também nos oferece a função `alert`, que mostra o dado que passamos como um “pop-up” de alerta ao usuário. No console, digite:

```
alert("Olá, mundo!");
```

Seguido de um Enter para que o navegador mostre, na aba associada ao console, o seguinte pop-up:



Fig. 3.2: alert olá mundo

Você já deve ter percebido que precisamos escrever o texto `Olá, mundo!` entre aspas. A linguagem requer que as usemos para diferenciar variáveis de valores de texto; veremos mais sobre este tópico no próximo capítulo.

Lendo dados da entrada

Assim como é possível escrever dados para uma saída, é possível ler dados de uma entrada. A entrada é, normalmente, o teclado do computador. Quando um programa pede à um usuário que este entre com algum dado, pode ser qualquer coisa: um texto, um número, um *sim* ou *não*, um nome de arquivo, uma data; o que quer que seja que o programa precisa, ele pode pedir ao usuário.

JavaScript nos oferece duas funções para receber dados dos usuários: `prompt` e `confirm`. `confirm` é a mais simples delas: passamos um texto, que é o que o usuário vê na tela, e o usuário apenas confirma ou nega a afirmação ou pergunta. É desta forma que fazemos perguntas “sim ou não” em JavaScript. Testemos:

```
confirm("Você gosta dos cursos da 4Linux?");
```

O código acima nos mostra o seguinte pop-up:



Fig. 3.3: confirm

E, ao clicar em OK, recebemos o valor booleano `true` (*verdadeiro*) de volta:

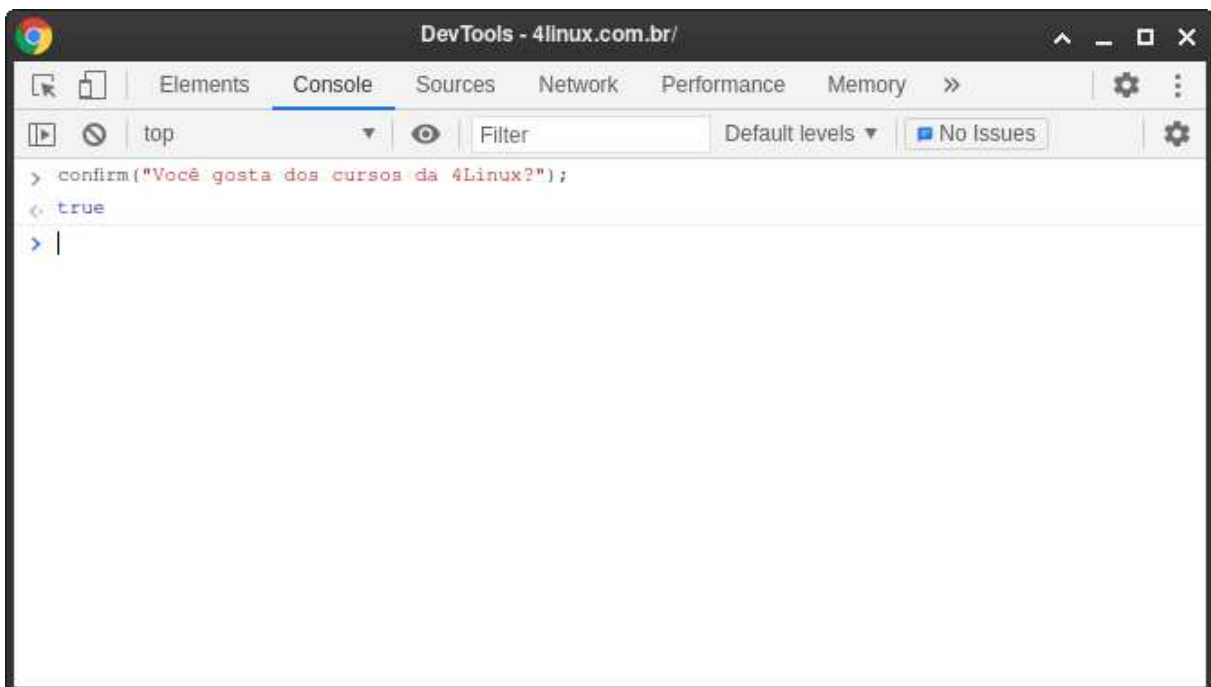


Fig. 3.4: confirm ok true

Analogamente, se clicarmos em Cancelar, recebemos o booleano `false` (*falso*):

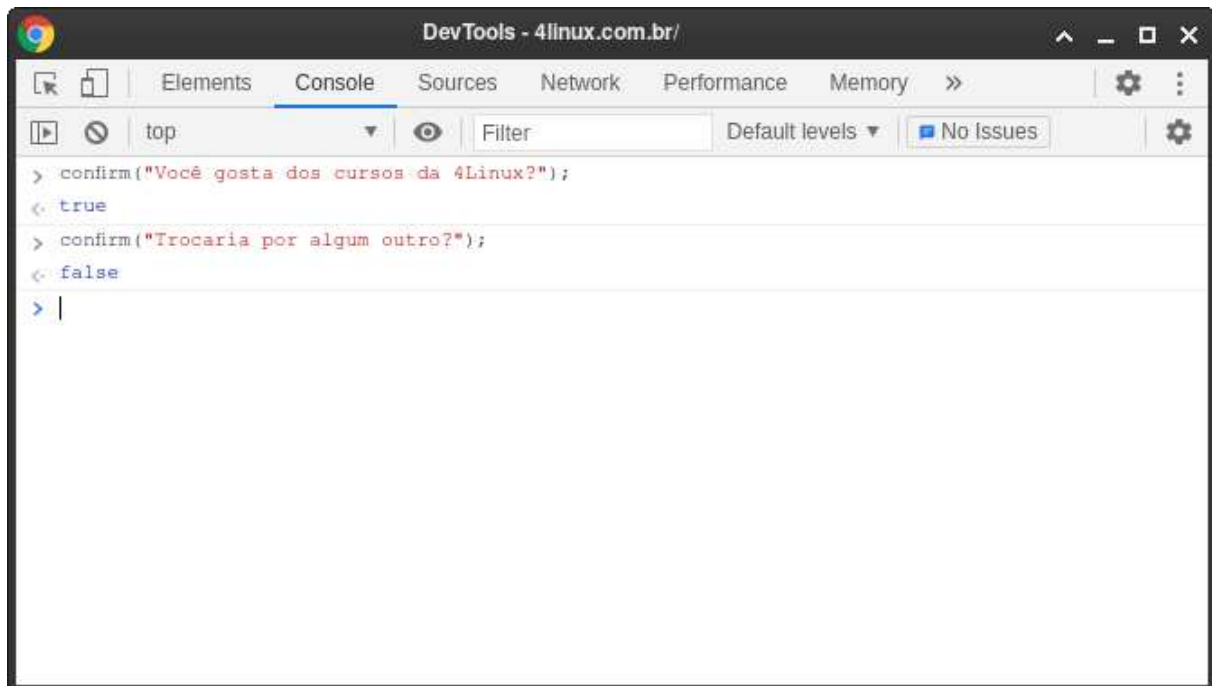


Fig. 3.5: confirm cancel false

Porém, nem sempre podemos contar com um simples `true` ou `false`. E se quisermos saber qual é a linguagem preferida do usuário? Neste caso, `prompt` se encaixa melhor nas nossas necessidades:

```
prompt("Qual é a sua linguagem preferida?");
```

A função nos devolve exatamente o que for escrito no pop-up:

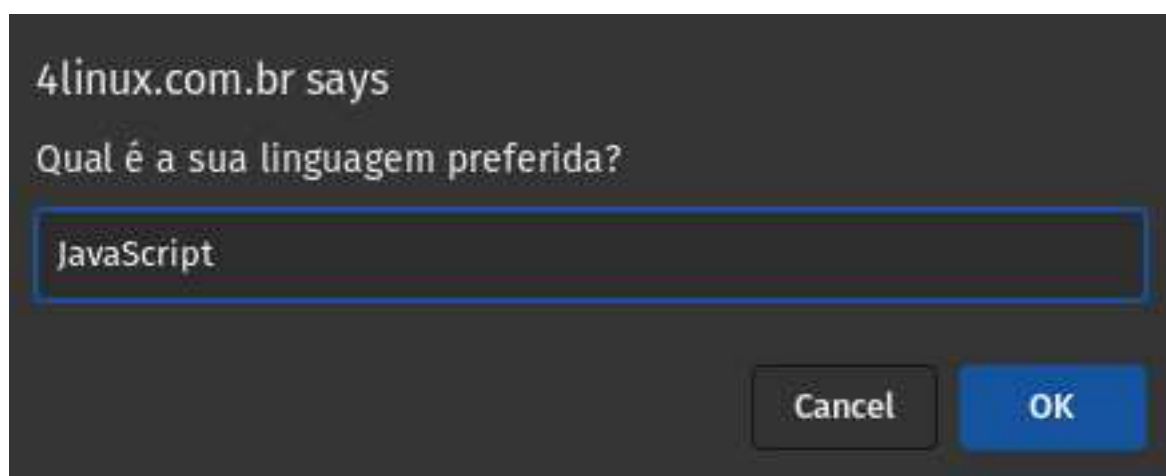


Fig. 3.6: prompt



Fig. 3.7: prompt answer

Além disso, `prompt` ainda nos permite fornecer um valor padrão:

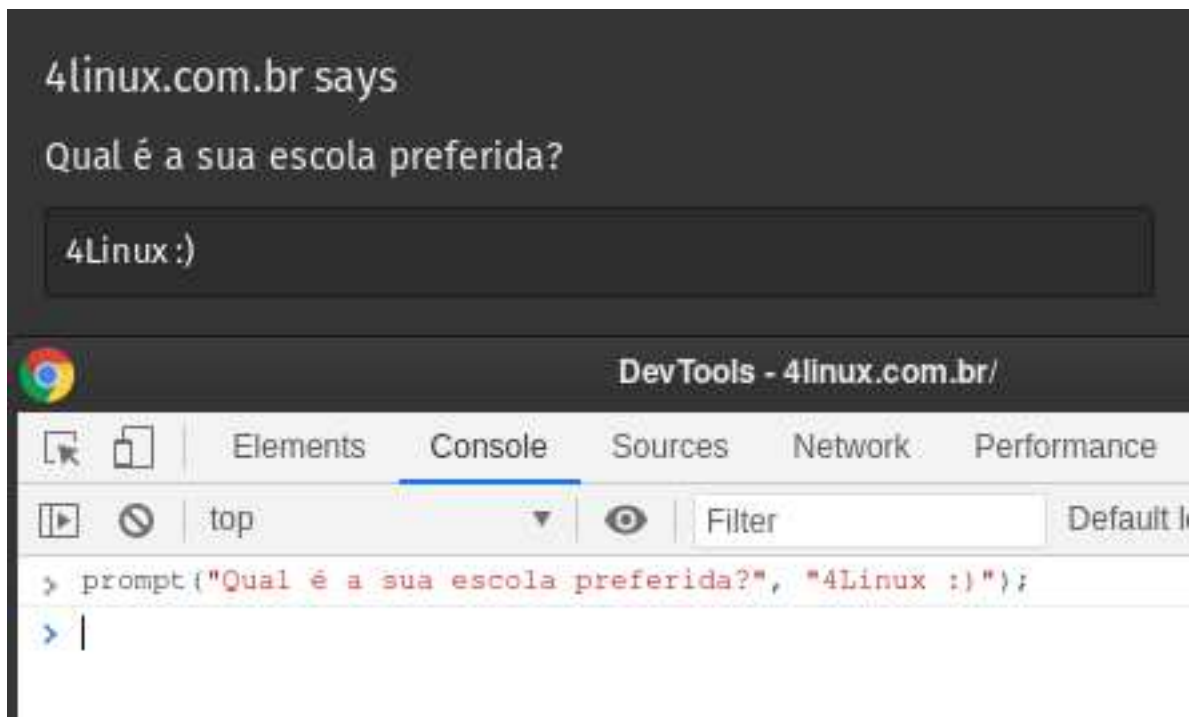


Fig. 3.8: prompt default value

4

Variáveis

O que é uma variável?

Uma variável, do ponto de vista do programador, é um nome ou identificador ao qual podemos atribuir um valor. Na matemática, a variável que mais vemos é x . Na programação, podemos não apenas ter x , como também `nome`, `idade`, `sacola` (para sacola de compras em um e-commerce, por exemplo), `alunos` (para uma lista de chamada de uma turma de escola, por exemplo). A “regra” é que usemos nomes descritivos para as nossas variáveis. x pode fazer sentido em um contexto matemático, porém pode ser melhor usar outros nomes em outros contextos.

Já do ponto de vista do computador, uma variável é um bloco de memória física de tamanho definido que pode armazenar certos tipos de dados - tipos estes que também têm tamanhos definidos. Algumas linguagens de programação diferenciam tipos numéricos de 1, 2, 4 e 8 bytes, a fim de salvar memória quando o número não é muito alto ou possibilitar números mais altos, fornecendo blocos de memória maiores.

Nomes válidos

Linguagens de programação possuem *palavras reservadas*, isto é, certas palavras que não podem ser usadas como identificadores porque a linguagem as dá um significado especial. Um exemplo é `if` (“se” em inglês); a palavra `if` é usada em estruturas de decisão e, por isso, não é um nome de variável válido. A validade de um nome pode variar de linguagem para linguagem:

em C e Java, `int` é um tipo de dado e, portanto, uma palavra reservada. Mas em JavaScript não há essa distinção; `int` pode ser usado como nome de variável.

Tipos primitivos

Tipos primitivos são tipos de dados que podem ser usados por tamanhos específicos de blocos de memória e não possuem atributos ou comportamentos atrelados a eles. A única coisa que os compõe é o próprio valor. Os tipos numéricos de 1 à 8 bytes mencionados anteriormente são exemplos de tipos primitivos. A linguagem C possui os seguintes tipos primitivos:

Tipo	Tamanho	Com ou sem sinal	Menor valor	Maior valor
<code>char</code> OU <code>int8</code>	8 bits (1 byte)	Com	-128	127
<code>unsigned char</code> OU <code>uint8</code>	8 bits (1 byte)	Sem	0	255
<code>short</code> OU <code>int16</code>	16 bits (2 bytes)	Com	-32768	32767
<code>unsigned short</code> OU <code>uint16</code>	16 bits (2 bytes)	Sem	0	65535
<code>int</code> OU <code>int32</code>	32 bits (4 bytes)	Com	-2,147,483,648	2,147,483,647
<code>unsigned int</code> OU <code>uint32</code>	32 bits (4 bytes)	Sem	0	4,294,967,295
<code>long</code> OU <code>int64</code>	64 bits (8 bytes)	Com	-9,223,372,036,854,775,808 9,223,372,036,854,775,807	9,223,372,036,854,775,807
<code>unsigned long</code> OU <code>uint64</code>	64 bits (8 bytes)	Sem	0	18,446,744,073,709,551,615

Os tipos numéricos assinalados usam 1 dos bits para representar todos os números negativos.

Algumas linguagens, como Python e JavaScript, não expõem tipos primitivos diretamente. Em vez disso, elas abstraem esses tipos em algo que chamamos de *objetos*. Objetos possuem atributos e comportamentos que vão além de um mero valor numérico.

Alguns tipos nativos do JavaScript

Linguagens de programação, por vezes, possuem interpretações diferentes dos tipos nativos. Em JavaScript, não existem vários tipos numéricos como os da tabela acima; nela há apenas `Number`. Também não existe um tipo para representar caracteres únicos como `char` do C, porém há as `Strings`, que são sequências de texto. Também há os `Arrays`, ou listas, e os `objetos` (`Object`

), que podem ter atributos e comportamentos e funcionam como mapeamentos chave:valor.

Tipo	Exemplos
Number	2, 3.14, Infinity, NaN (N ot a N umber), -7
String	"a", "texto", 'aspas simples'
Array	[], [1, 2, 3], ['tipos diferentes', true, Infinity]
Object	{}, {nome: "João", idade: 30}

No exemplo de Object acima, nome e idade são chaves e "João" e 30 são os seus respectivos valores.

Operadores aritméticos

No desenvolvimento de software, operadores aritméticos funcionam como na matemática. Afinal, uma das coisas que mais se faz em programas é conta. Podemos fazer somas, subtrações, multiplicações, divisões e restos normalmente usando os seguintes operadores:

Operador	Nome	Descrição opcional	Exemplo
+	Adição		2 + 3
-	Subtração		7 - 4
*	Multiplicação		3 * 5
/	Divisão		14 / 7
%	Módulo	Calcula o resto da divisão dos operandos	11 % 3

Usando JavaScript para demonstrar na prática

Em uma janela de navegador (Mozilla Firefox, Google Chrome, etc), acesse as ferramentas de desenvolvedor (Chrome: F12; Firefox: Ctrl (ou Cmd) + Shift + J) e, na aba *Console*, digite algumas operações matemáticas para ver seus resultados.

Alguns exemplos:

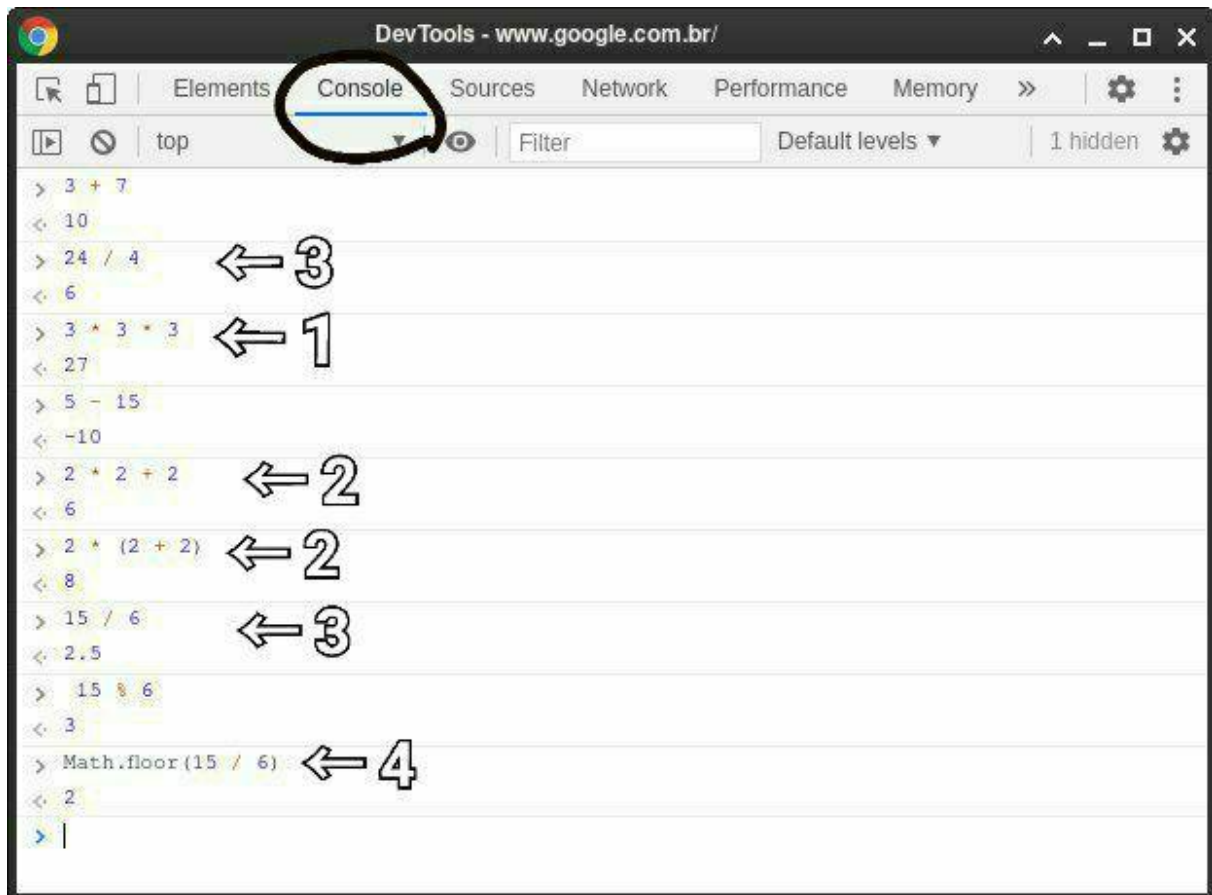


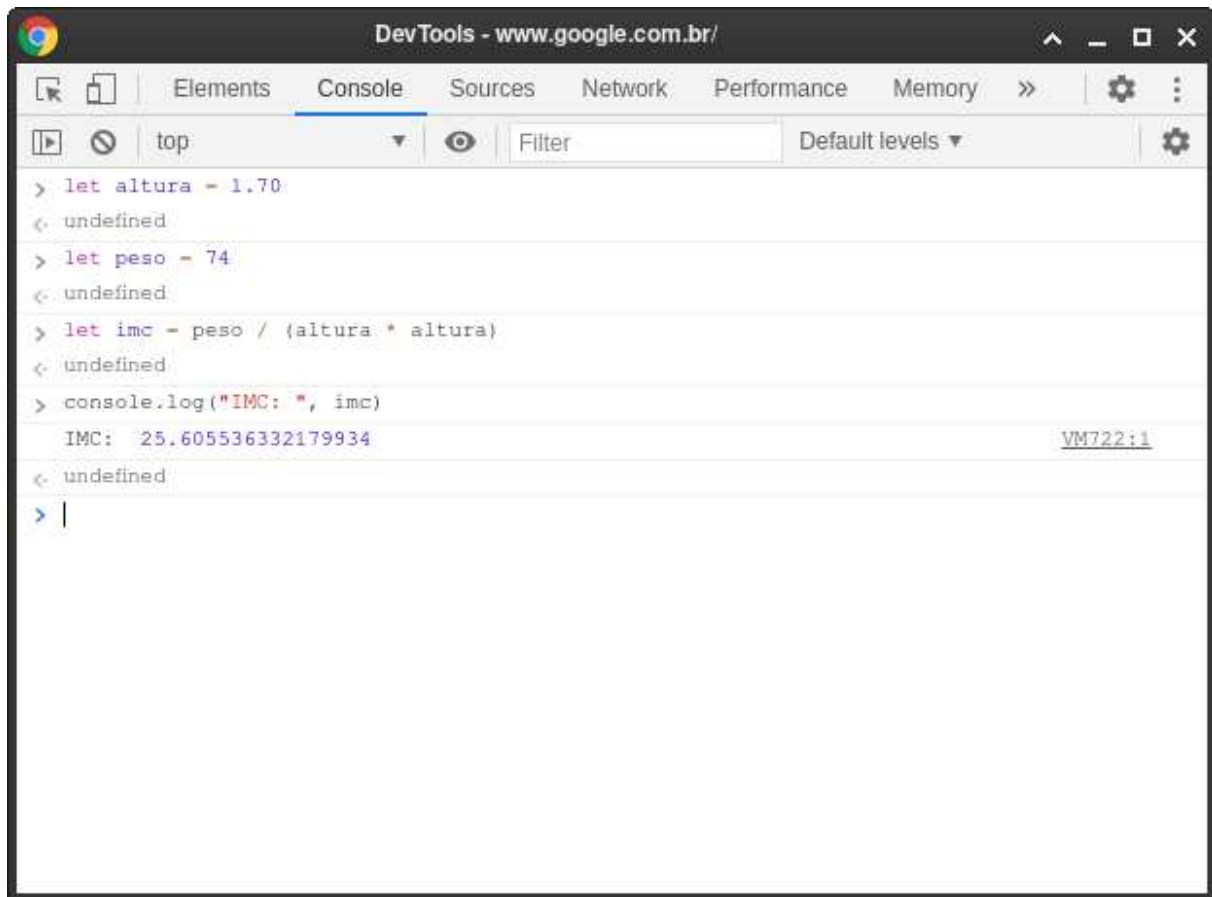
Fig. 4.1: chrome::console::arithmetic

Perceba que:

1. É possível fazer mais de uma operação em uma só expressão.
2. Os operadores têm uma ordem de precedência assim como na matemática.
 - E parênteses podem remover ambiguidades ou “alterar” a precedência.
3. Divisões podem resultar em um número inteiro ou de ponto flutuante (com casas decimais).
4. `Math.floor(n)` pode ser usado para transformar o número `n` em inteiro.

E usando variáveis...

...para calcular IMC:

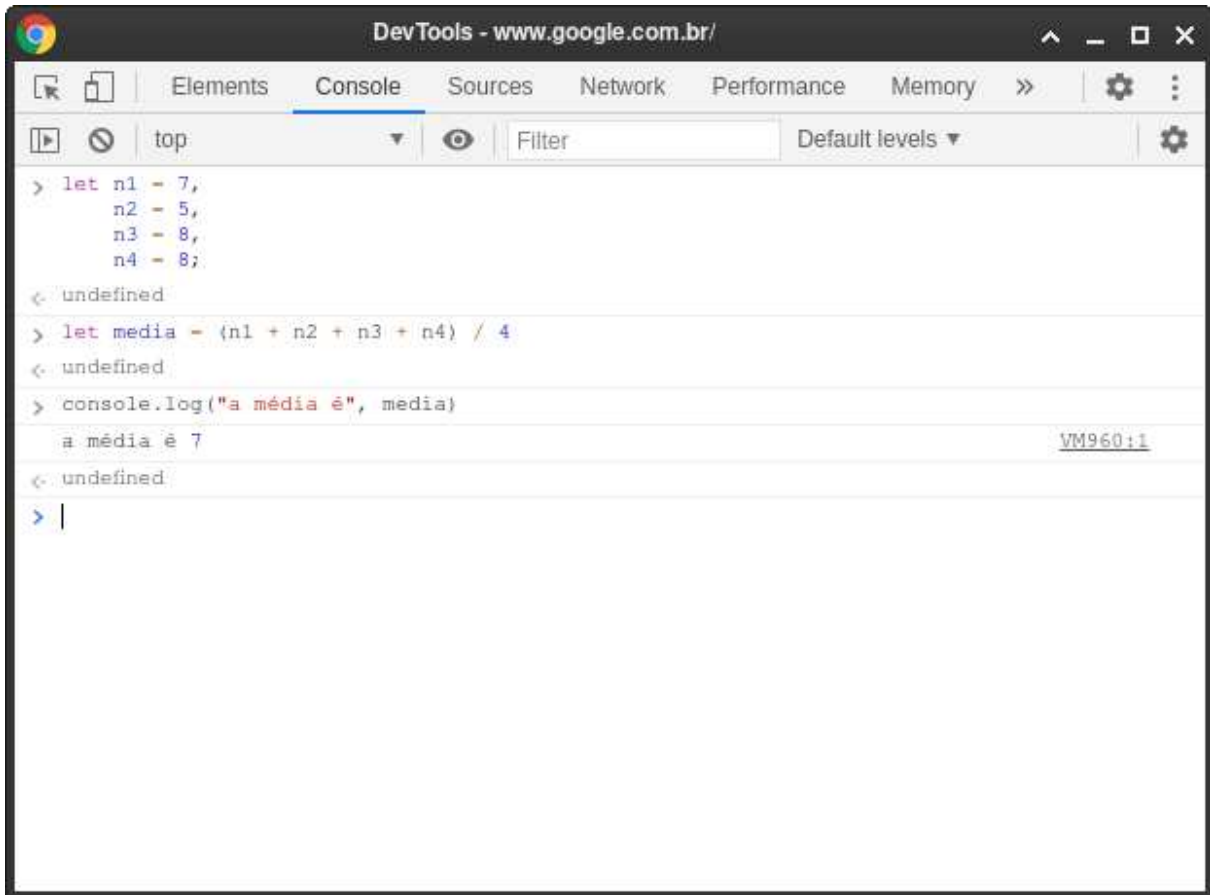


The screenshot shows the Chrome DevTools Console with the following code and output:

```
> let altura = 1.70
< undefined
> let peso = 74
< undefined
> let imc = peso / (altura * altura)
< undefined
> console.log("IMC: ", imc)
IMC:  25.605536332179934
< undefined
> |
```

The output shows the BMI calculation result: 25.605536332179934. The console also shows the source file path: VM722:1.

... para calcular média de notas escolares:



Exercícios

1. Abra o console. Declare algumas variáveis e defina seus valores. Brinque com os operadores; use parênteses, misture operações. Tente fazer operações com outros tipos, como somar strings, por exemplo. Comece com:

```
let x = 3, y = 8;
let nome = "seu nome", sobrenome = "seu sobrenome";
```

2. Declare e defina arrays. Acesse os elementos individualmente. Tente acessar índices inválidos (como -1, por exemplo). Comece com:

```
let lista = ['a', 2, 3.14];
console.log(lista[2]);
```

3. Declare e defina objetos. Crie e exclua atributos de um objeto criado. Comece com:

```
let pessoa = {nome: "João", idade: 30};
```

```
    pessoa.nascimento = 1991;  
    delete pessoa.idade;
```

5

Estrutura de decisão

Se / senão

Um dos aspectos mais importantes no mundo do desenvolvimento de software é a habilidade de tomar decisões. Devo mostrar a página com as vendas da loja para o usuário logado? A secretária deve ter acesso aos documentos que a médica escreve sobre os pacientes? O assinante do plano tem acesso à esse canal de TV? Para todas essas perguntas, o software precisa conseguir decidir, de acordo com os dados que ele possui, se ele segue o caminho da esquerda ou da direita. Por isso, linguagens de programação possuem a estrutura se senão para tomada de decisões.

Essencialmente, funciona da seguinte forma:

```
se alguma condição for verdadeira {  
    fazer algo para a condição verdadeira  
}  
caso contrário {  
    fazer algo para a condição falsa  
}
```

Vale destacar que a cláusula do caso contrário (*senão*) é opcional e pode ser omitida, se não for necessária.

Usando os mesmos exemplos da primeira aula, podemos abrir a janela de ferramentas de

desenvolvedor do navegador e digitar:

```
let euDirijo = true;

if (euDirijo) {
  console.log("Posso trabalhar com algo que envolva dirigir.");
}

euDirijo = !euDirijo;

if (!euDirijo) {
  console.log("Não sei dirigir, então não preciso de um carro.");
}
```

Ao executarmos, temos:



Como euDirijo possui o valor verdadeiro (true) e é a única variável usada na cláusula se (if), a condição é verdadeira e o bloco de código entre as chaves é executado. Logo depois, dizemos que euDirijo = !euDirijo; o ponto-de-exclamação (!) é um operador lógico em JavaScript que serve para negar um valor ou expressão booleana. É como se escrevêssemos euDirijo = !true

e, como a negação de true é false, euDirijo passa a ter o valor falso. Então, no próximo if, a verificação passa, pois estamos validando a negação de um valor; a negação do valor falso é o verdadeiro.

```
let maiorDeIdade = true,
    habilitado = true;
let possoDirigir;

if (maiorDeIdade && habilitado) {
    possoDirigir = true;
}
else {
    possoDirigir = false;
}

if (possoDirigir) {
    console.log("Sou maior de idade e habilitado. Posso dirigir.");
}
```

No exemplo acima, usamos dois valores e a conjunção para definirmos se podemos dirigir:



Fig. 5.1: chrome::console::if::2

Mude os valores de `maiorDeIdade` e `habilitado` e teste para ver o que acontece.

Se encadeado

Às vezes, um mero `if / else` não é o suficiente, e precisamos fazer uso de mais do que apenas uma cláusula `if` em uma instrução de tomada de decisão. Por isso, a linguagem nos possibilita encadear as instruções, fazendo com que tenhamos uma cadeia com um controle mais granular.

Considere o seguinte pedaço de código:

```
let n = 35;

if (n % 3 === 0) {
  console.log(n, "é divisível por 3.");
}
else if (n % 5 === 0) {
  console.log(n, "é divisível por 5.");
}
else if (n % 7 === 0) {
  console.log(n, "é divisível por 7.");
}
else {
  console.log(n, "não é divisível por 3, 5 ou 7.");
}

console.log("após a cadeia");
```

35 é um número divisível tanto por 5 quanto por 7, mas ao executarmos o código:

```

> let n = 35;

if (n % 3 === 0) {
  console.log(n, "é divisível por 3.");
}
else if (n % 5 === 0) {
  console.log(n, "é divisível por 5.");
}
else if (n % 7 === 0) {
  console.log(n, "é divisível por 7.");
}
else {
  console.log(n, "não é divisível por 3, 5 ou 7.");
}

console.log("após a cadeia");
35 "é divisível por 5."
após a cadeia
undefined

```

Em nenhum momento somos informados de que o número *também* é divisível por 7. Isto ocorre porque assim que uma condição é validada como verdadeira, o programa interrompe a cadeia de ifs e pula diretamente para a primeira instrução depois da cadeia.

Operadores Lógicos

O JavaScript possui alguns operadores lógicos que podemos usar para realizar validações mais precisas. Os operadores servem para que possamos fazer conjunções, disjunções e negações.

Operador	Descrição	Exemplo	Resultado
!	negação	!false	true
&&	conjunção	true && false	false
	disjunção	true false	true

Operadores de comparação

Além dos operadores lógicos, temos também os operadores de comparação, que são os operadores que nos dão valores booleanos que podemos usar em nossos ifs. Os operadores são:

Operador	Descrição	Exemplo	Resultado
===	Igualdade	1 === 1	true
!==	Diferença	9 !== 9	false
<	Menor que	8 < 8	false
<=	Menor ou igual a	50 <= 50	true
>	Maior que	15 > 15	false
>=	Maior ou igual a	21 >= 18	true

Note também que, porque essas operações resultam em um valor booleano, é possível atribuir o resultado da expressão a uma variável:

```
let maioridadeLegal = 18,
    idade = 20;

let maiorDeIdade = idade >= maioridadeLegal;

console.log(maiorDeIdade);

let habilitado = true;

let podeDirigir = maiorDeIdade && habilitado;

console.log(podeDirigir);
```

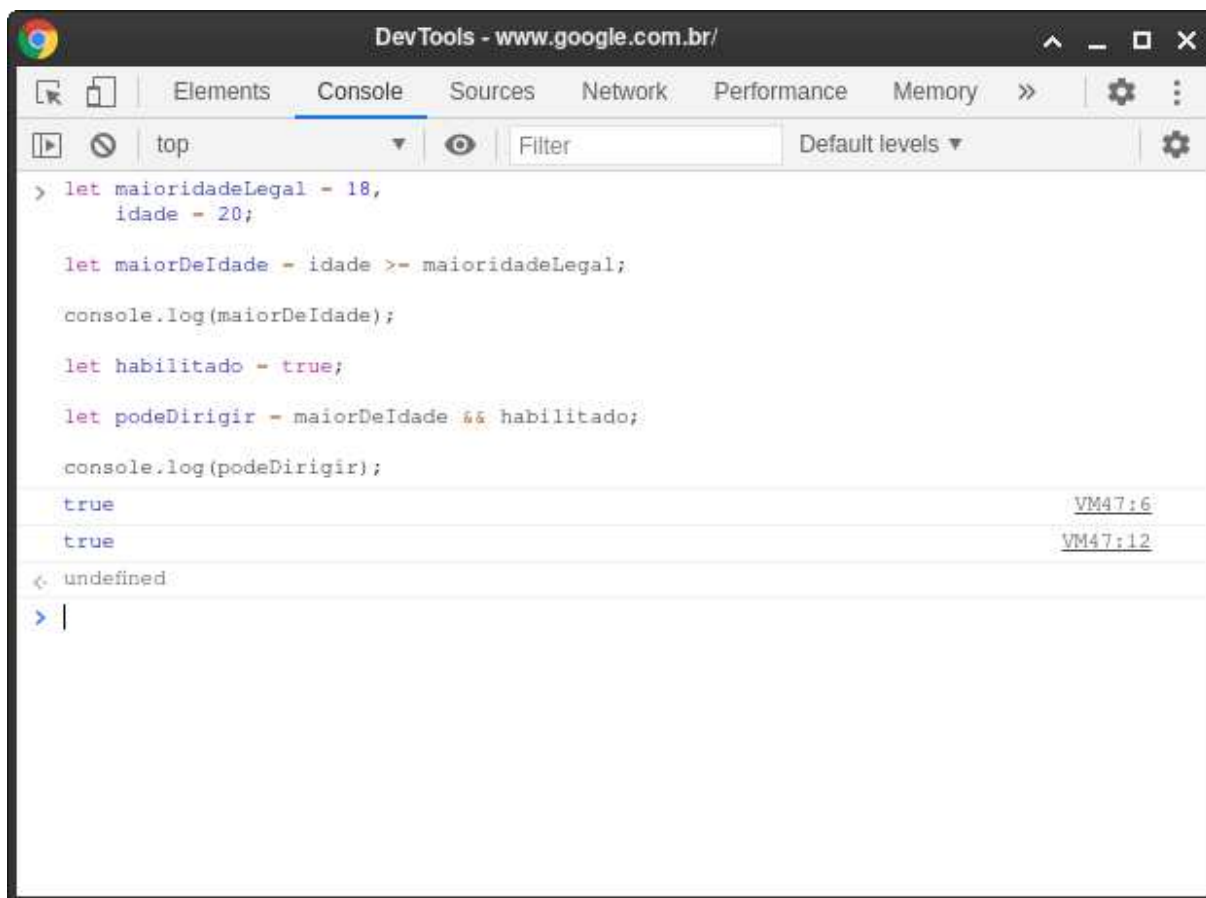


Fig. 5.2: chrome::console::cmp

Operador ternário de decisão

Também existe uma estrutura mais enxuta de tomada de decisão que é o que chamamos de operador ternário. É possível encontrar referências à este operador com o nome de “operador Elvis”, pela forma como se parece: `?:`. A sintaxe das expressões que usam este operador é:

```
(expressão booleana) ? (valor caso verdadeiro) : (valor caso falso)
```

Então considerando novamente o exemplo de dirigir:

```

let idade = 28,
    habilitado = true;

if (idade >= 18 && habilitado) {
  console.log("eu dirijo");
}

```

```
else {  
    console.log("eu não dirijo");  
}
```

Pode ser reescrito como:

```
let idade = 28,  
    habilitado = true;  
  
console.log(idade >= 18 && habilitado ? "eu dirijo" : "eu não dirijo");
```

Ou até mesmo:

```
let idade = 28,  
    habilitado = true;  
  
console.log("eu" + (idade >= 18 && habilitado ? " " : " não ") + "dirijo");
```

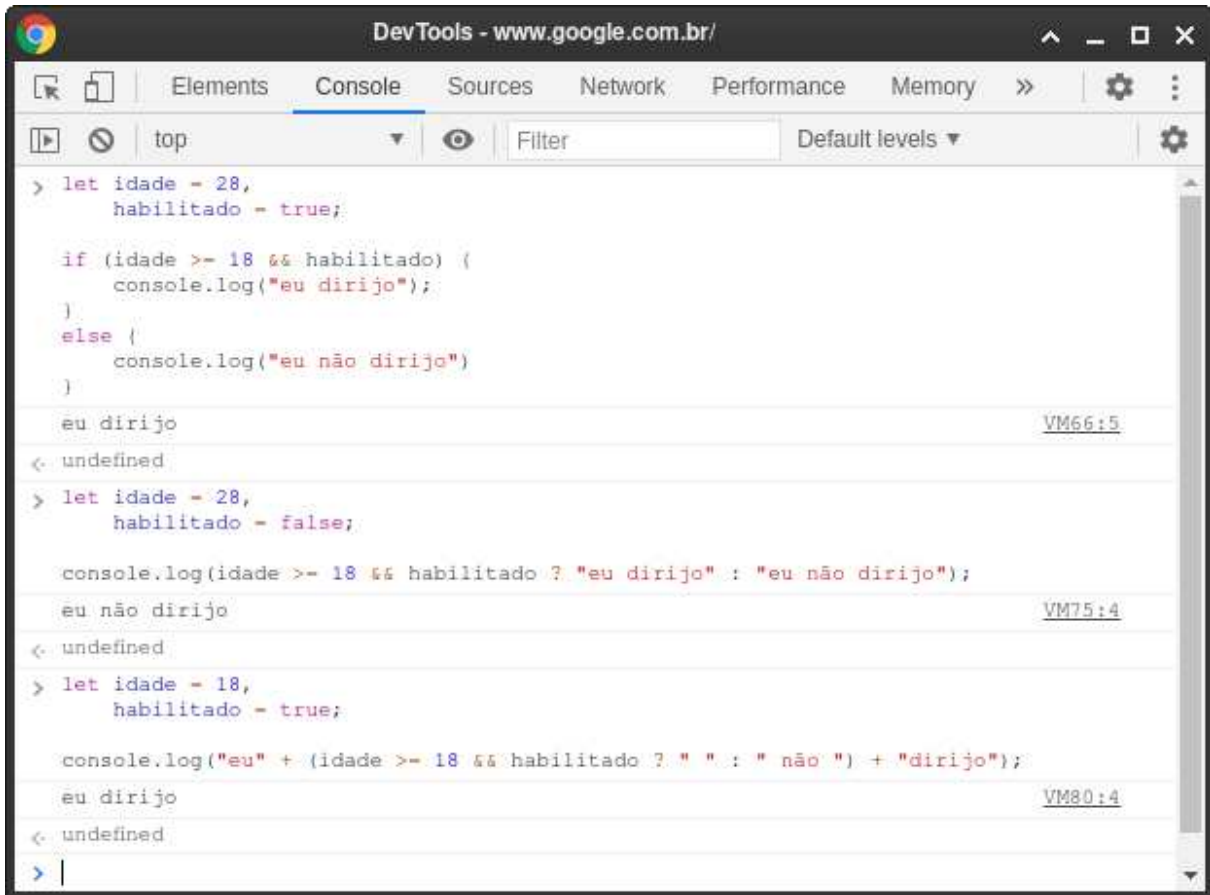


Fig. 5.3: chrome::console::elvis

Exercícios

1. Escreva um programa que exiba uma mensagem de acordo com a hora atual:

- Boa madrugada se a hora estiver entre 0 e 5;
- Bom dia se estiver entre 6 e 11;
- Boa tarde se estiver entre 12 e 17;
- Boa noite caso contrário.

A hora atual pode ser definida por você no código. Certifique-se via código de que o valor da hora esteja sempre entre 0 e 23.

```

let hora = 0,
    msg;

msg = ...;
console.log(msg);

```

2. Refaça o exercício anterior, porém usando apenas operadores ternários em vez de cadeias

de ifs.

6

Estruturas de repetição

Às vezes, é necessário que um conjunto de instruções seja executado repetidamente. As repetições podem ocorrer até que uma condição seja atendida: um certo número de vezes ou até mesmo para todos os elementos de uma sequência. Nesta aula, veremos as duas primeiras formas de repetição com os comandos `while` e `for`.

Comando `while`

O comando `while` (“enquanto” em inglês) serve para iniciar um bloco de código que se repetirá *enquanto* uma condição for *verdadeira*. Um exemplo simples é dividir um número por 2 até que o resultado da divisão seja ímpar:

```
let n = 1000;

while (n % 2 === 0) {
  n = n / 2;
}

console.log(n);
```

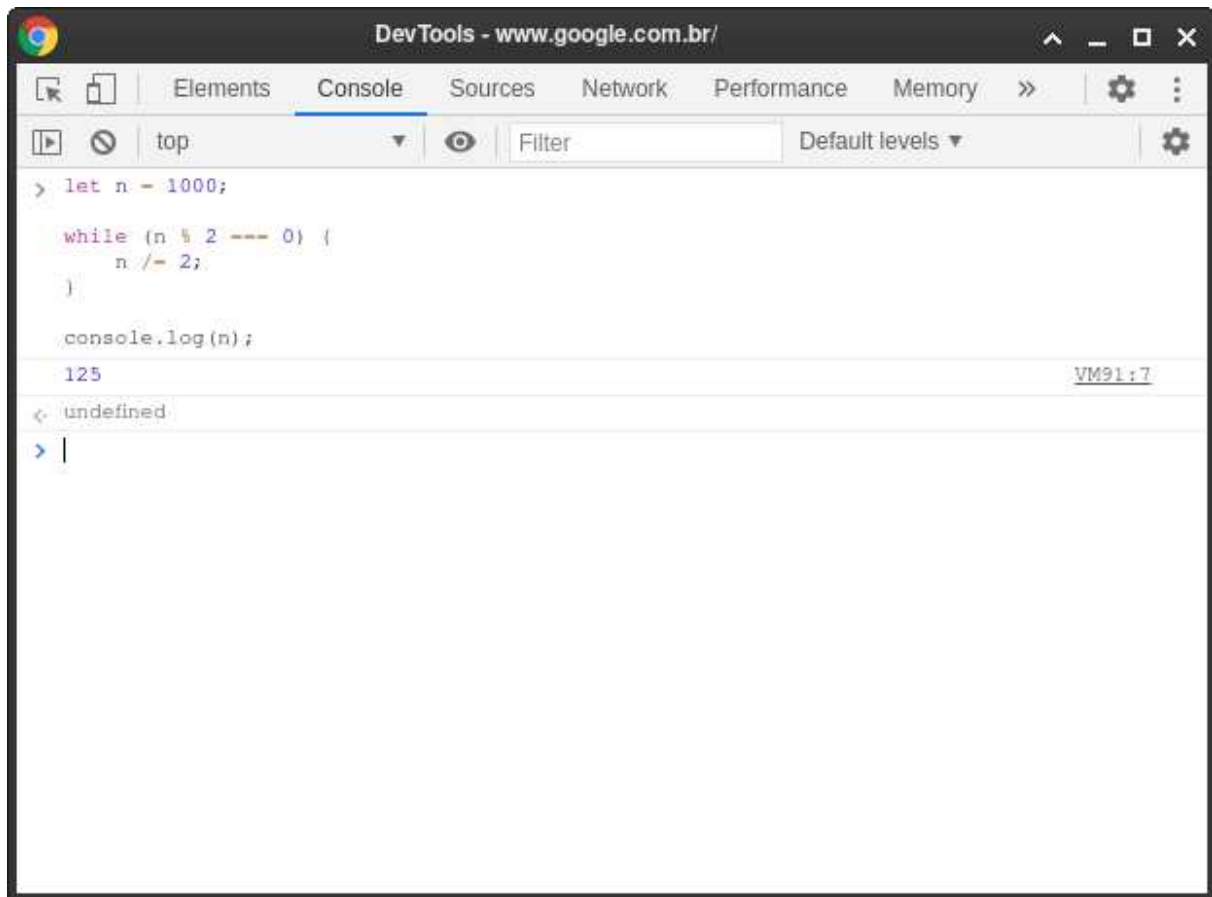


Fig. 6.1: chrome::console::while

As iterações (voltas) do loop acontecem como na tabela abaixo:

Iteração	n	n é par? ($n \% 2 === 0$)	n após a divisão por 2
1	1000	true	500
2	500	true	250
3	250	true	125
4	125	false	

Ou seja, no momento em que n, 125, é testado novamente no início do loop, o programa valida que o valor não é par e o interrompe, pulando para a primeira instrução logo após o bloco do loop.

Note que nenhuma instrução do loop será executada se a expressão da condição for falsa desde o começo:

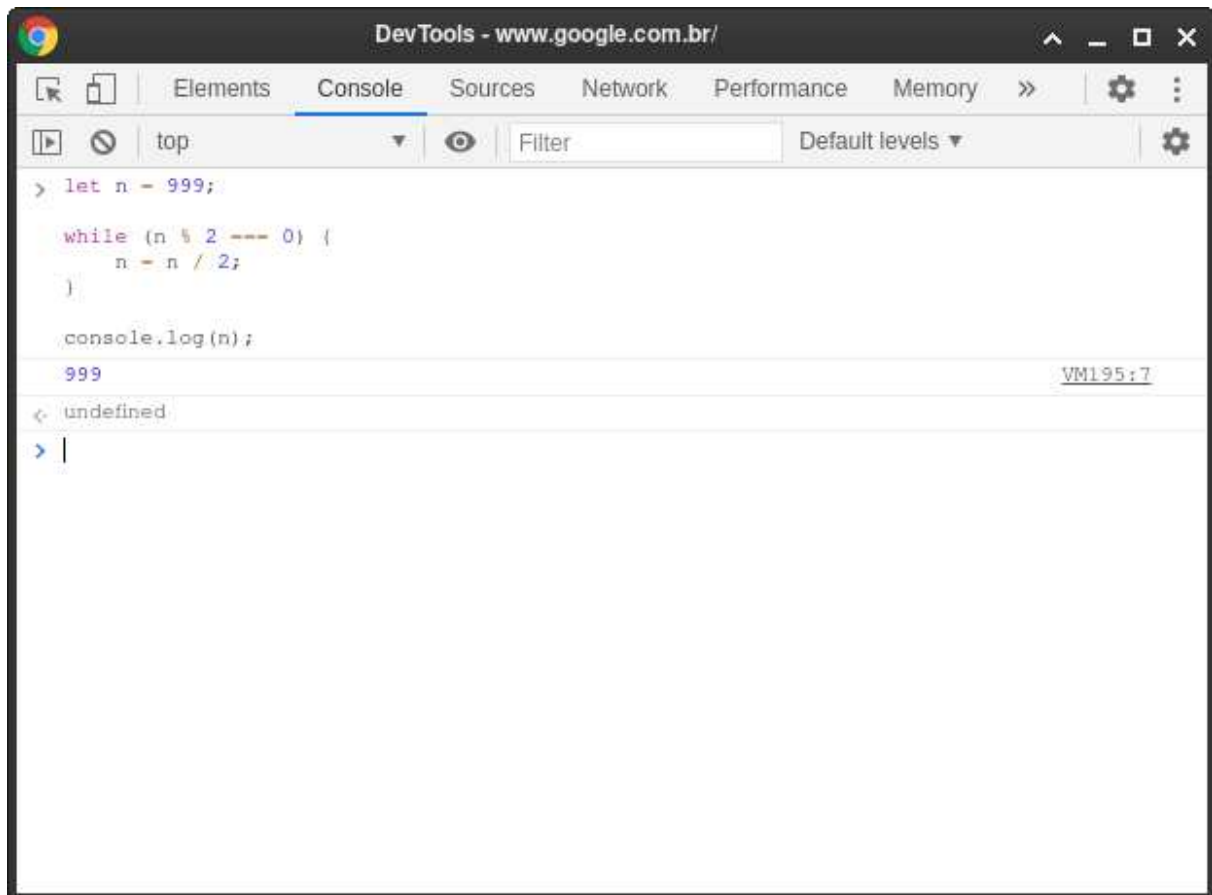


Fig. 6.2: chrome::console::while-false

Com `while` também podemos fazer o que chamamos de *loop infinito*: o loop executa enquanto a condição for verdadeira. Mas e se a condição sempre for verdadeira?

```

let n = 1;

while (true) {
  console.log(n++);
}
  
```

Aviso: o código acima travará o console de desenvolvedor e pode até travar o seu navegador inteiro.

do ... while

Foi mencionado acima que o código do loop não será executado caso a condição seja falsa desde o começo, mas a linguagem possui um constructo que garante a primeira execução em

qualquer caso: `do {} while ()` (“faça ... enquanto” em inglês). A diferença sintática entre esse constructo e o anterior é meramente a adição da palavra `do` e o deslocamento da palavra `while` para depois das chaves:

```
let n = 1001;

do {
  n = n / 2;
} while (n % 2 === 0);

console.log(n);
```

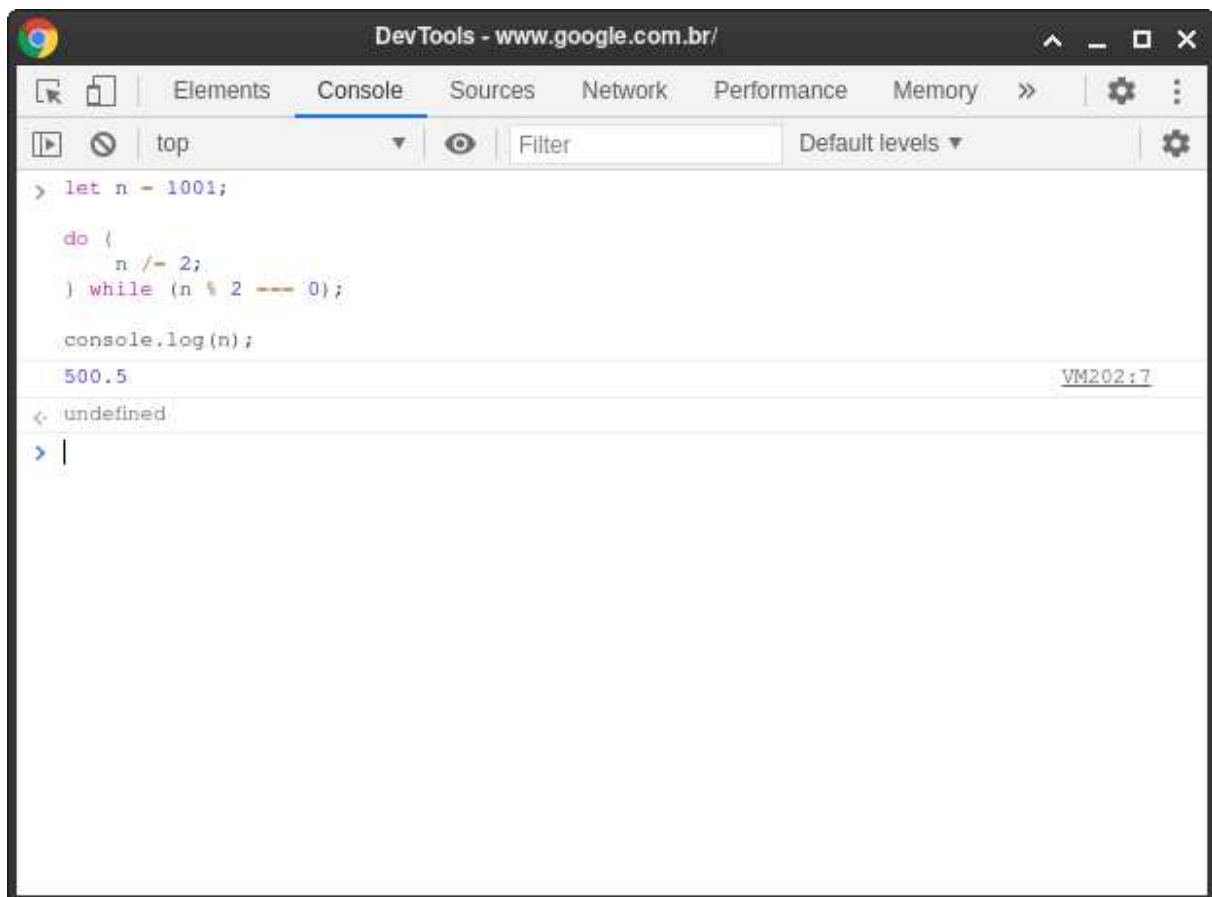


Fig. 6.3: chrome::console::do-while

Comando for

O comando `for` (“para” em inglês) possui o mesmo fim do comando `while`, ou seja, executar um conjunto de instruções repetidamente. O que muda é o meio: usamos `for`, normalmente,

quando já sabemos quantas vezes queremos repetir o bloco de código. Ele é composto por três partes em sua essência:

```
for (inicialização; condição; expressão)
```

Onde:

- inicialização pode conter variáveis que desejamos inicializar;
- condição pode conter a condição para que o loop continue executando;
- expressão pode conter uma expressão que será executada no final de cada iteração.

As três partes são opcionais e é possível fazer loops infinitos com `for` escrevendo apenas `for (;;)`.

Para contar de 1 a 10 usando um loop `for`, podemos escrever:

```
for (let i = 1; i <= 10; i++) {  
  console.log(i);  
}
```

Ou, para fazer duas contagens ao mesmo tempo, de 1 a 10 e de 9 a 0:

```
for (let i = 1, j = 9; i <= 10 && j >= 0; i++, j--) {  
  console.log(i, j);  
}
```

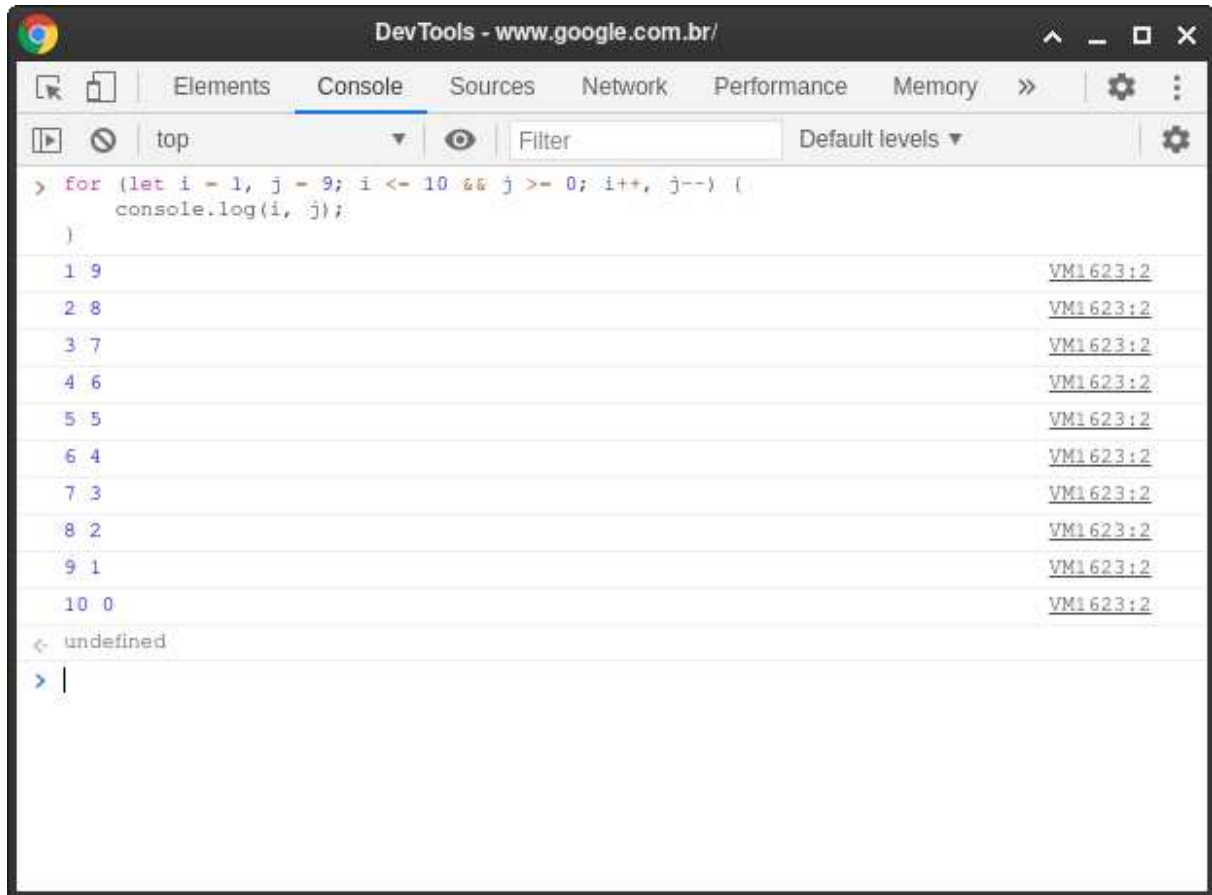


Fig. 6.4: chrome::console::for

O mesmo loop escrito com `while`:

```
let i = 1,
    j = 9;

while (i <= 10 && j >= 0) {
  console.log(i, j);
  i++;
  j--;
}
```

Controlando o fluxo

Além do controle básico que as estruturas de loop provêm, a linguagem ainda oferece duas outras formas de controlar o fluxo de um loop. Uma delas é pulando para a próxima iteração do loop antes de chegar ao fim do bloco de código; outra é interrompendo completamente o

loop e seguindo diretamente para a primeira instrução após o bloco. Aqui temos os comandos `continue` e `break`.

Um exemplo real de uso do `break` seria quando tentamos encontrar algum conteúdo interessante na TV. Começamos por um canal específico e avançamos até que encontremos algo que nos interesse ou a lista de canais acabe. Neste segundo caso, desligamos a TV e vamos fazer alguma outra coisa.

Supondo canais de 1 a 100, podemos escrever em pseudo-código:

```
canal = 1
enquanto (canal <= 100) {
    se (conteúdo é interessante) {
        break
    }

    incrementar canal em 1
}

se (canal é 101) {
    desligar a TV
}
```

Fazemos a última checagem, porque o número do canal só será 101 se nós passarmos por todos os 100 canais e não encontrarmos algo para assistir. Se o canal 42 estiver apresentando um conteúdo interessante, podemos parar de procurar e nos estabelecer no sofá.

Já, para exemplificar o `continue`, podemos pensar em um molho de chaves com diferentes tipos de chaves e uma fechadura.



Fig. 6.5: diferentes tipos de chaves

Nós não precisamos *testar* as chaves do tipo 1 ou 2 em uma fechadura do tipo 3 porque *sabemos* que não vai entrar. Podemos simplesmente pular estas chaves, ou *continuar* a

procurar a chave desejada no molho. Em pseudo-código:

```
tipo de fechadura = 3
para cada chave no molho de chave {
    se (tipo da chave não é igual ao tipo de fechadura) {
        continue
    }

    tentar colocar chave na fechadura

    se (chave entrou) {
        tentar girar a chave na fechadura
        se (fechadura girou) {
            trancar ou destrancar a porta
            break
        }
    }
}
```

Exercícios

1. Usando um loop `for`, escreva um programa que calcula o fatorial de um número `n`.
 - $n! = n * (n-1)!$
 - $1! = 1$
2. Escreva um programa que mostra de 1 a 100 no console, exceto os divisíveis por 3 ou 5; o programa deve escrever `Fizz` para números divisíveis por 3, `Buzz` para números divisíveis por 5 e `FizzBuzz` para números divisíveis por ambos.

7

Funções

Escrever código dia após dia se torna cansativo com o tempo. Principalmente quando precisamos reescrever código que já havíamos escrito anteriormente. Escrevemos, escrevemos e percebemos que poderíamos reusar boa parte dele, mudando apenas uma coisa ou outra aqui e ali. Com isso, surgiu o conceito de blocos de código reutilizáveis e parametrizáveis: blocos de código que são genéricos o suficiente para permitir que sejam usados em uma miríade de contextos com poucas alterações. Estas são o que chamamos de funções nas linguagens de programação atuais.

Declaração / Chamada de uma função

Em boa parte dos casos, funções têm um *nome*, uma *lista de parâmetros* e um *bloco de código* associados à elas. Para declararmos uma função em JavaScript, usamos a palavra reservada `function` seguida de um par de parênteses, que é onde ficam os parâmetros; seguido de um par de chaves com o código dentro. Considere uma função que nos mostra no console se um número é par ou ímpar:

```
function parOuImpar(numero) {  
  let palavra = numero % 2 === 0 ? "par" : "ímpar";  
  console.log(numero + " é " + palavra);  
}
```

Temos:

- o nome da função, `parOuImpar`,
- o único parâmetro, `numero`, e
- o bloco de código entre as chaves.

Perceba que, no bloco de código, você pode criar variáveis e executar instruções normalmente. As variáveis criadas dentro da função com `let` serão excluídas quando a execução chegar ao fim da função.

Quando chamamos (usamos) uma função, precisamos passar *argumentos* a ela. Os argumentos são os valores que os parâmetros assumem. No caso da nossa função, o argumento pode ser qualquer número:

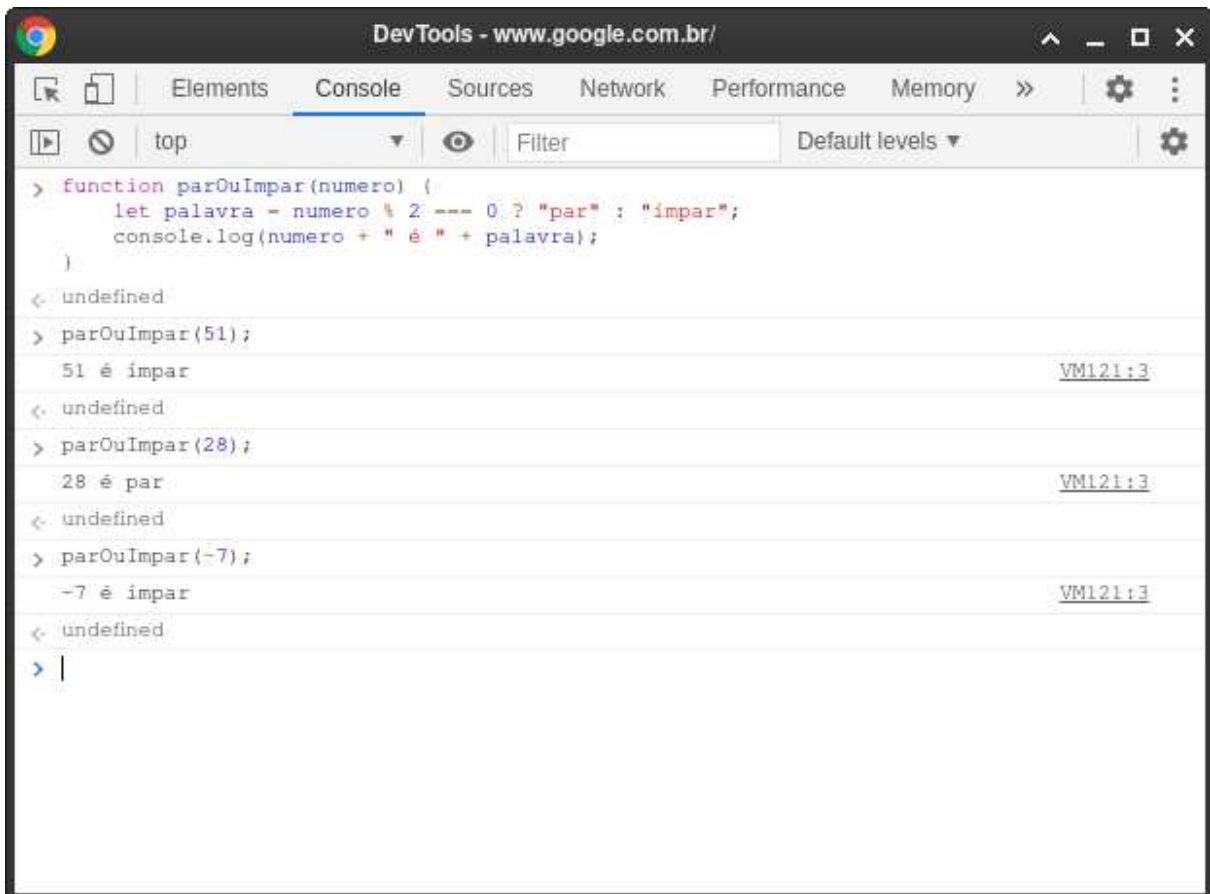


Fig. 7.1: definição e chamada de função

Desta forma, definimos um bloco de código que pode ser reutilizado indefinidamente e é adaptável às nossas necessidades - no caso, podemos usar com qualquer número.

Passagem de parâmetros

Toda função aceita um número indefinido de parâmetros - de zero à indefinido. De fato, existem maneiras de manipular um número indeterminado de argumentos por constructos que a linguagem nos oferece.

Funções sem nenhum parâmetro

Funções com zero parâmetros são funções que não necessitam de nenhum valor “externo” para funcionar. Elas trabalham com o que têm. Um exemplo pode ser uma função que mostra um menu no console:

```
function mostrarMenu() {  
  let opcoes = ["Entrar", "Criar conta", "Sair"];  
  
  console.log("Digite a opção desejada");  
  for (let i = 0; i < opcoes.length; i++) {  
    console.log(i + 1 + " " + opcoes[i]);  
  }  
}
```

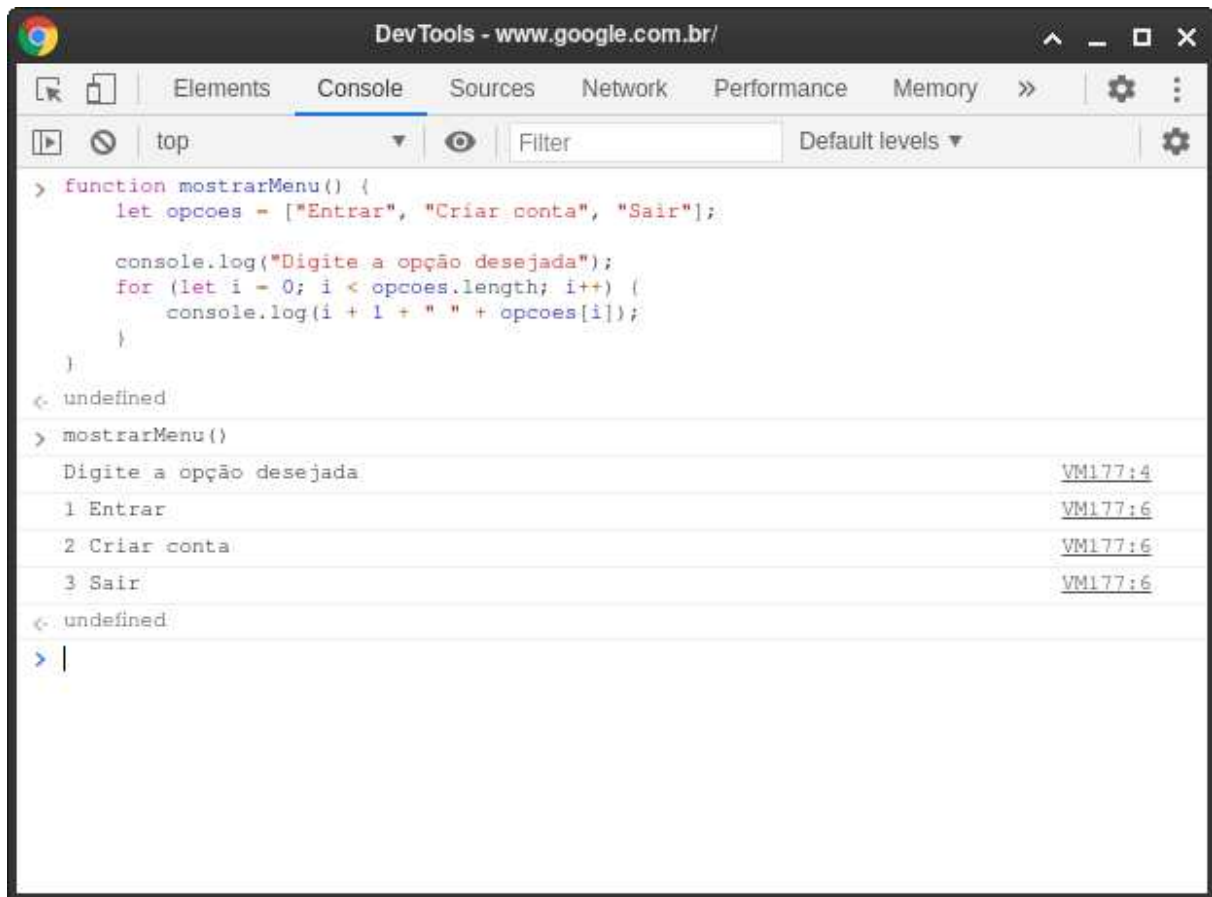


Fig. 7.2: criando e chamando mostrarMenu

Funções com um ou mais parâmetros

A princípio, pode parecer que nossa função de mostrar um menu é o suficiente para o que ela foi feita - e, de fato, pode ser em algumas ocasiões. Mas e se o nosso sistema tiver mais de um menu e precisarmos mostrar cada um deles de acordo com um fluxo? Faremos uma função para cada menu?

Ainda que possamos, sim, criar uma função para cada menu, é muito mais prático reutilizar o que temos, deixando as partes que mudam entre uma chamada e outra parametrizáveis. Podemos fazer com que a lista de opções do menu seja um parâmetro da função como a seguir:

```
function mostrarMenu(opcoes) {
  console.log("Digite a opção desejada");
  for (let i = 0; i < opcoes.length; i++) {
    console.log(i + 1 + " " + opcoes[i]);
  }
}
```

Ou até mesmo controlar de qual número iniciar a contagem:

```
function mostrarMenu(opcoes, numeroInicial) {
  console.log("Digite a opção desejada");
  for (let i = 0, n = numeroInicial; i < opcoes.length; i++, n++) {
    console.log(n + ' ' + opcoes[i]);
  }
}
```



Fig. 7.3: parametrizando mostrarMenu com opcoes e numeroInicial

Retorno

Assim como podemos passar dados (argumentos) às funções, elas também podem nos devolver, ou retornar, outros dados. Por exemplo, faz sentido que uma função cujo nome é `somarNumeros` aceite uma lista de números como entrada e devolva a soma deles. Assim funciona a entrada e saída de funções. Em JavaScript:

```
function somarNumeros(numeros) {  
  let soma = 0;  
  
  for (let i = 0; i < numeros.length; i++) {  
    soma += numeros[i];  
  }  
  
  return soma;  
}
```

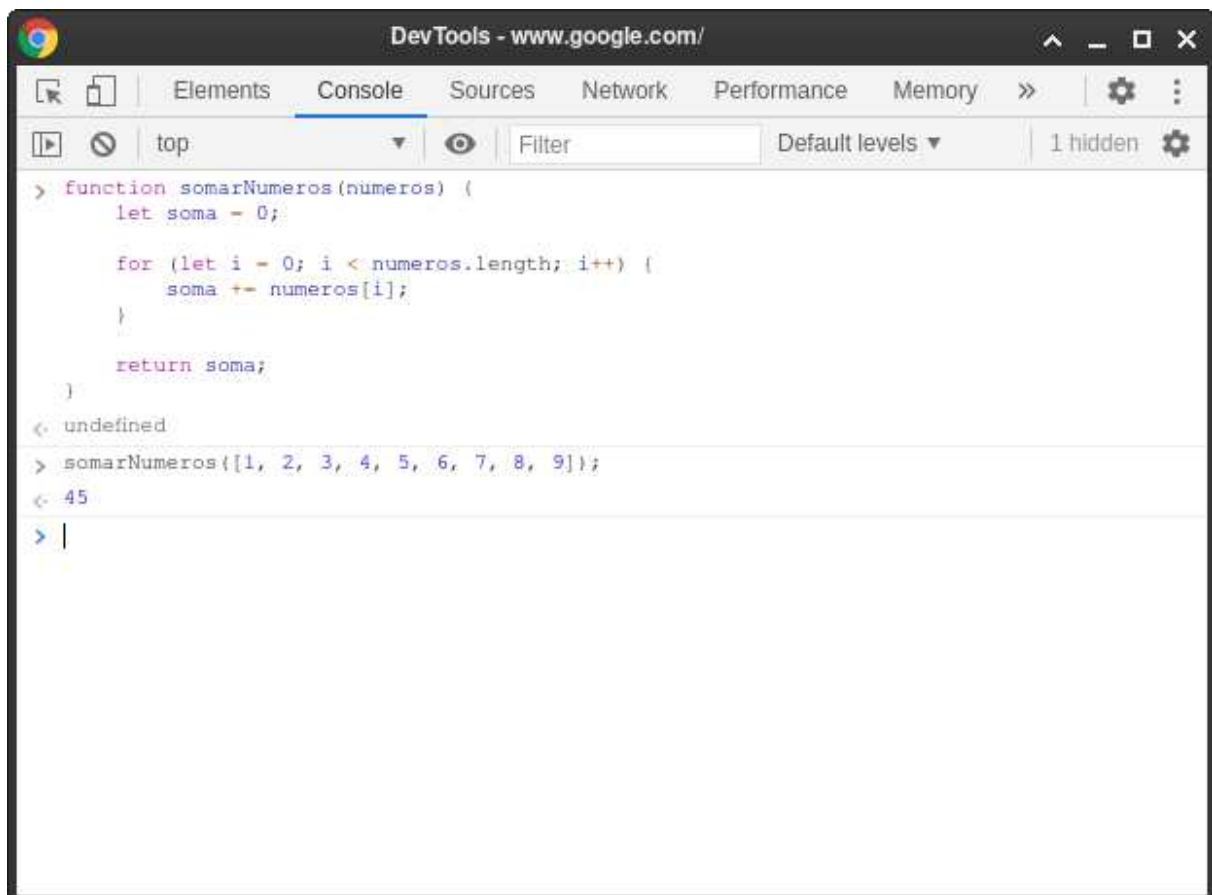


Fig. 7.4: funções retornando valores

Os valores que as funções retornam podem ser atribuídos a variáveis e usados em expressões subsequentes:

```
let soma = somarNumeros(7, 8, 8, 9);  
let media = soma / 4;  
console.log("A média é " + media);  
console.log(somarNumeros(5, 5, 10) / 3);
```

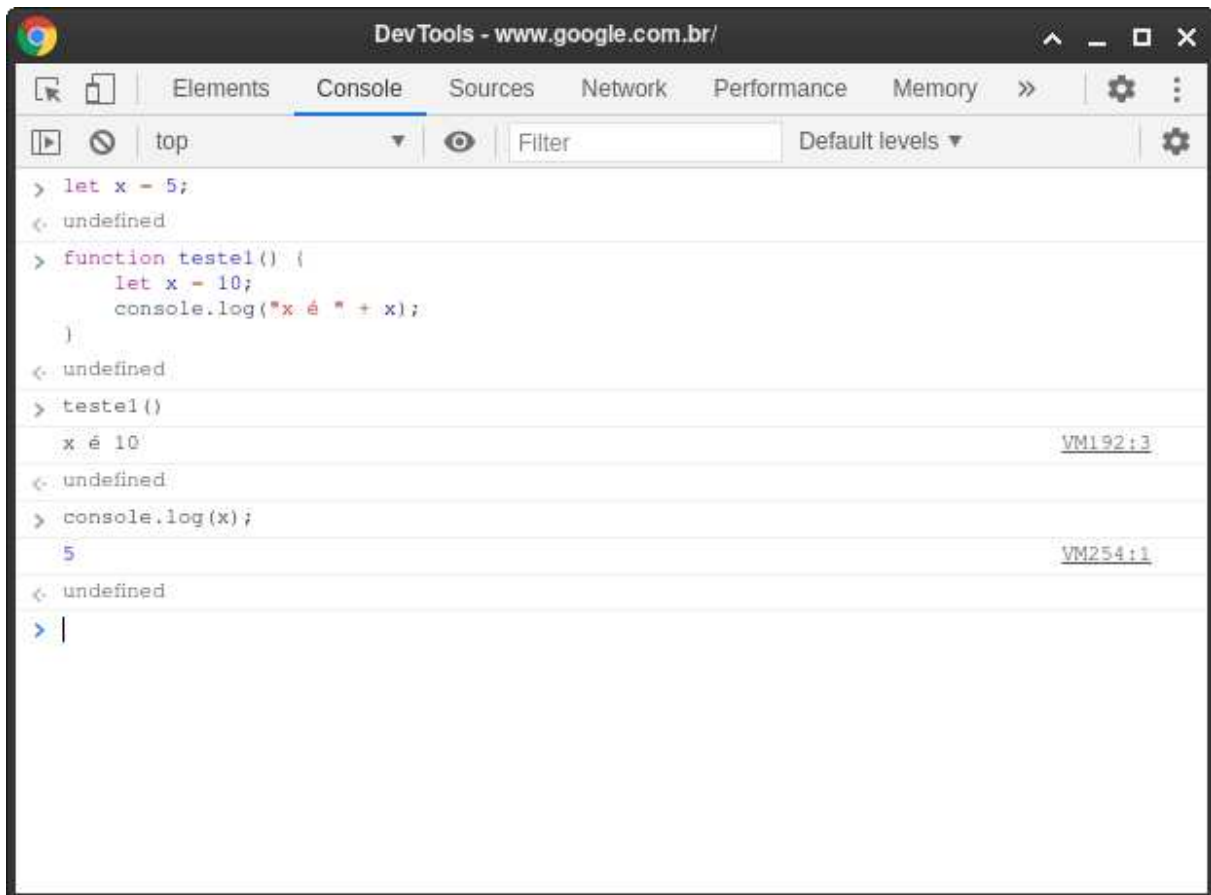
Escopo da função

Os blocos de código de funções possuem algo que chamamos de escopo. O escopo é o que contém variáveis *locais* e o ambiente que executa o código usa este escopo para saber quando variáveis existem e precisam de espaço na memória do computador. O ambiente também é responsável por apagar estas variáveis e liberar o espaço de memória utilizado por elas. Por este motivo, utilizamos a palavra reservada `let` em JavaScript para declarar variáveis: `let` faz com que as variáveis declaradas sejam atreladas ao bloco de código em que se encontram e excluídas logo depois da execução dele. Façamos alguns testes.

```
let x = 5;

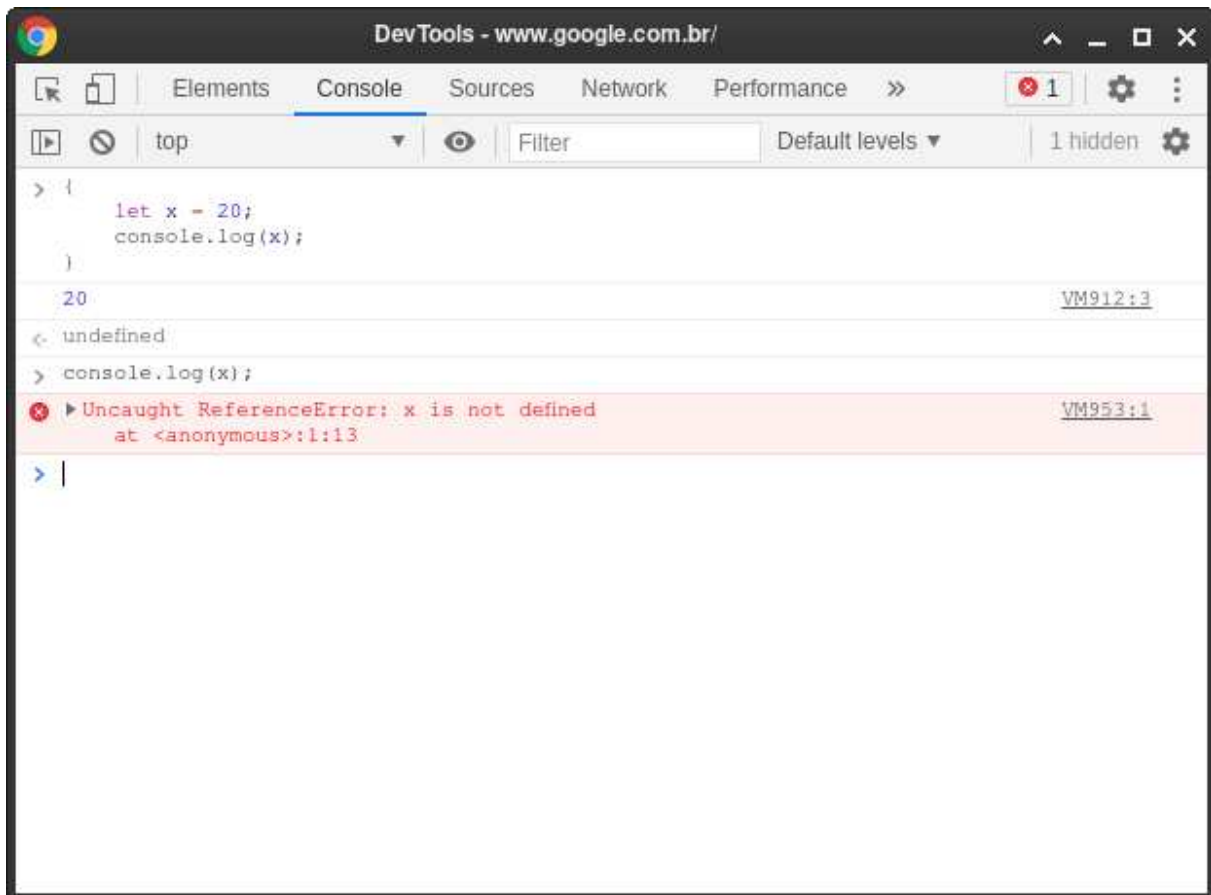
function teste1() {
  let x = 10;
  console.log("x é " + x);
}

teste1();
console.log(x);
```



Vemos que o `x` dentro da função não altera o `x` fora dela.

```
{  
  let x = 20;  
  console.log(x);  
}  
console.log(x);
```

Vemos que `x` só existe dentro do bloco de código delimitado por `{}`. Fora dele, ele não existe.

```

let x = 100;

function teste2() {
  x = 50;
  console.log('x é ' + x);
}

teste2();
console.log(x);
  
```

```

> let x = 100;

function teste2() {
  x = 50;
  console.log('x é ' + x);
}
< undefined
> teste2();
x é 50 VM1615:5
< undefined
> console.log(x)
50 VM1675:1
< undefined
> |

```

Vemos que, quando um valor é atribuído a `x` sem usar `let` dentro da função, o `x` alterado é o de fora; não é *criado* um novo `x` dentro da função, como nos exemplos anteriores.

Exercícios

1. Escreva uma função que calcula as raízes de equações de segundo grau usando a fórmula de bhaskara. O retorno deve ser um vetor com dois elementos, caso haja raízes (`[x1, x2]`), ou `null`, caso contrário. Use `Math.sqrt(numero)` para calcular a raiz quadrada de `numero`.

```

function resolveBhaskara(a, b, c) {
  return null;
}

```

2. Escreva uma função que calcula quanto de lucro uma empresa privada de transporte faz ao levar passageiros a certos destinos em uma viagem. Leve em conta que o gasto de gasolina varia de 150 a 300 por viagem e que cada destino tem um valor diferente. A entrada deve ser um vetor de destinos e deve conter no mínimo um e no máximo doze

destinos.

Suponha os seguintes preços para os destinos:

Destino	Preço
a	30
b	35
c	40
d	45

Exemplos:

Entrada	Gasto de gasolina	Saída
['a', 'b', 'c', 'd']	150	0
['a', 'a', 'b', 'd', 'a']	225	-55
['c', 'c']	300	-220
['a', 'a', 'b', 'b', 'c', 'c', 'd', 'd']	250	50

```
function calculaLucro(...) {
  let gastoGasolina = Math.random() * 150 + 150;

  return 0;
}
```

8

Vetores

Vetores são uma estrutura de dados essencial em programação. Essencialmente, eles representam uma sequência linear de valores indexados numericamente, onde 0 é o primeiro elemento e $n-1$ (para um vetor de tamanho n) é o último*. São comumente chamados de vetores, *arrays* ou listas.

* Algumas poucas linguagens de programação e ferramentas iniciam a indexação a partir de 1, porém o mais comum é a partir de 0.

Criação de Vetores

Para criar um vetor em JavaScript, usa-se colchetes. Um vetor vazio pode ser criado ao abrir e fechar um par de colchetes: `[]`. Também é possível criar um vetor com dados ao separá-los por vírgula dentro dos colchetes como, por exemplo: `[1, 2, 3]`.

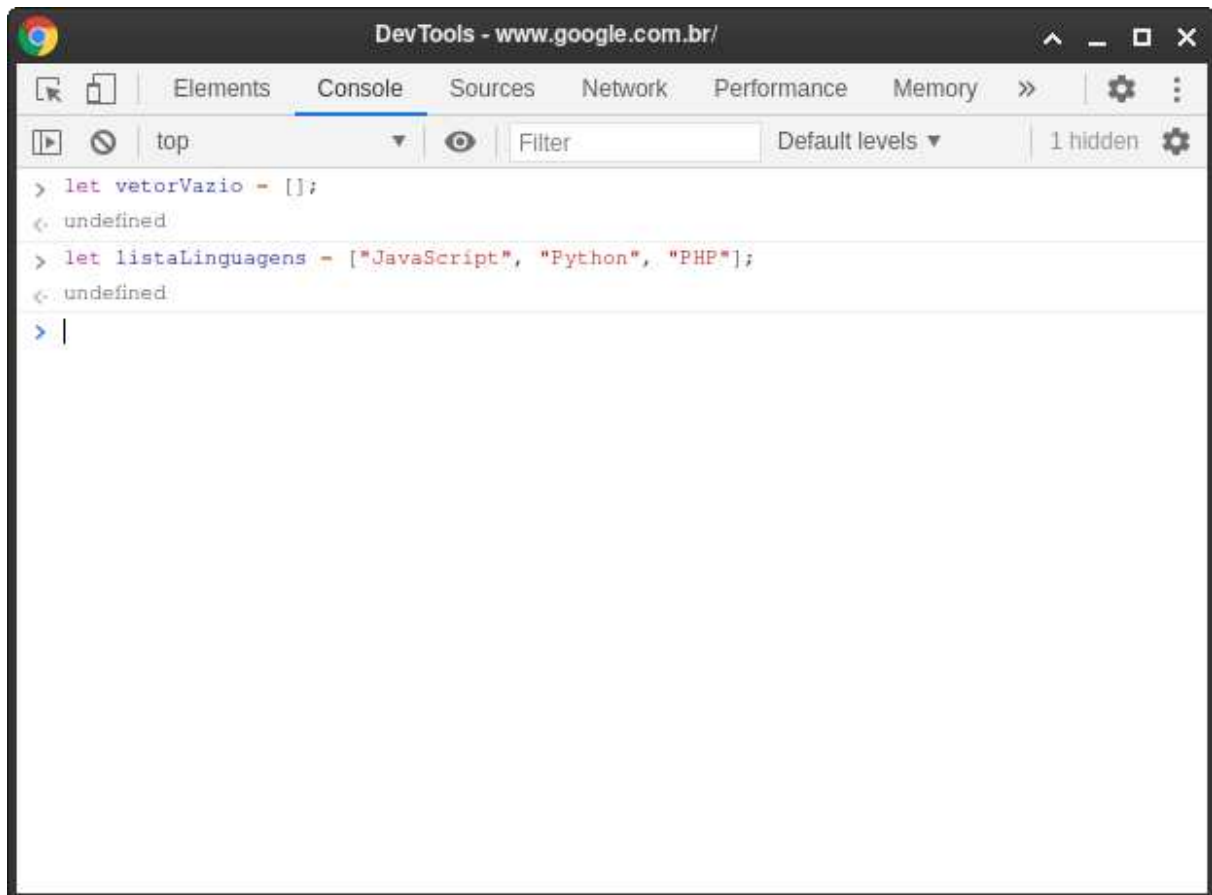


Fig. 8.1: criando vetores em js

Manipulação de Vetores

Vetores possuem comportamentos, funções associadas a eles (que chamamos de métodos) que podem ser usados para manipulá-los. As manipulações possíveis incluem:

- adicionar itens ao início ou ao final do vetor;
- remover itens do vetor;
- executar uma função para todos os elementos do vetor criando um segundo vetor a partir dele;
- filtrar um vetor de acordo com uma função predicado;
- entre outros.

Adicionando itens ao vetor

Existem dois métodos que podemos usar para adicionar itens aos nossos vetores. `unshift` adicionará um item à esquerda (no início), enquanto `push` adicionará um item à direita (no

fim).

```
let v = [1, 2, 3, 4];  
  
v.push(5);  
console.log(v);  
  
v.unshift(0);  
console.log(v);
```

Além desses, também há o método `concat`, que *concatena* dois vetores, ou seja, adiciona todos os itens de um segundo vetor ao final do primeiro vetor em um terceiro vetor (não altera os dois primeiros).

```
let v0 = [1, 2, 3],  
    v1 = [4, 5, 6];  
  
let v2 = v0.concat(v1);  
  
console.log(v0, v1, v2);
```

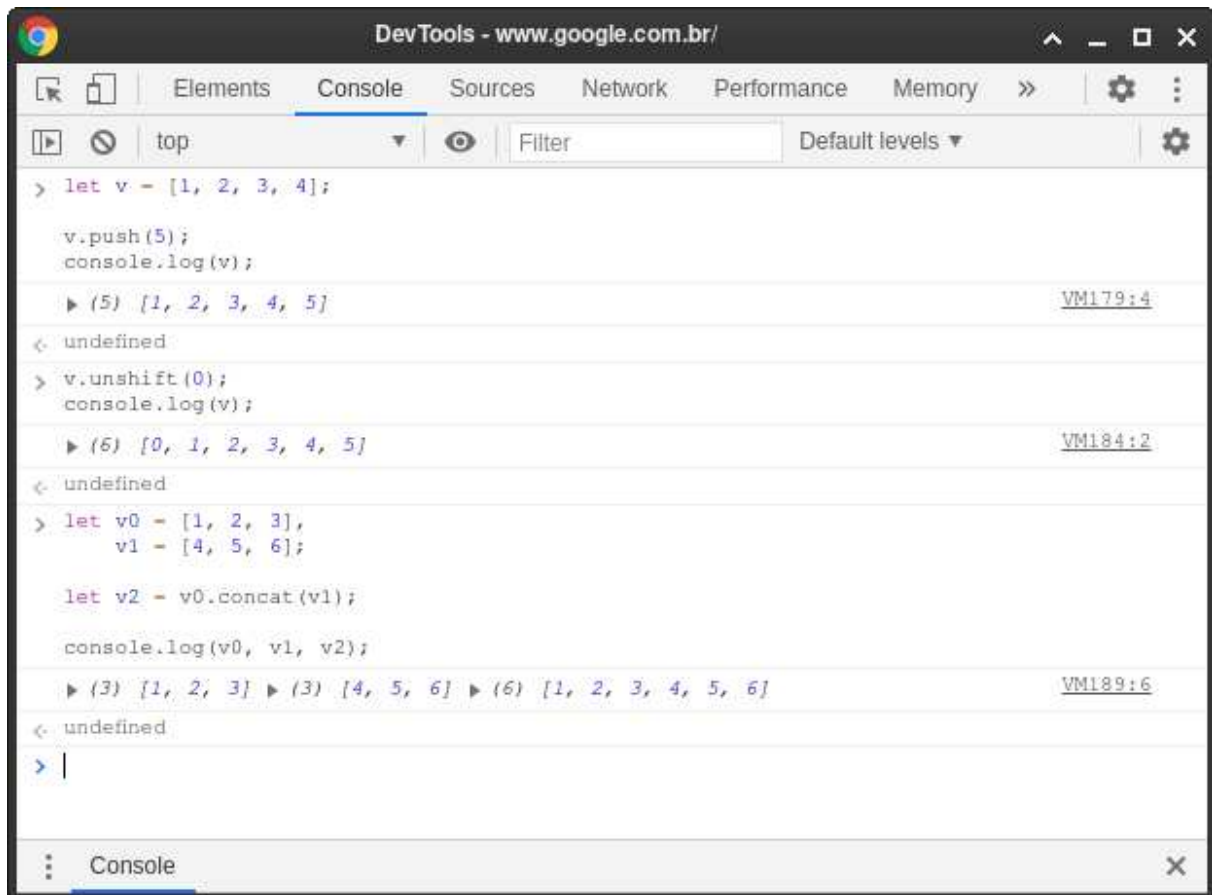


Fig. 8.2: adicionando itens em vetores

Removendo itens do vetor

Assim como unshift e push adicionam itens nas pontas do vetor, shift e pop removem itens delas. O item removido é devolvido pelo método.

```
let v = [0, 1, 2, 3, 4, 5];

let ultimo = v.pop(),
    primeiro = v.shift();

console.log(v);
console.log(primeiro, ultimo);
```

E podemos, também, remover um ou mais elementos a partir de qualquer posição do vetor com splice:

```
let elemsDoMeio = v.splice(1, 2);
console.log(elemsDoMeio);
console.log(v);
```

`v.splice(1, 2)` onde 1 é a posição de onde começar a tirar (a segunda posição) e 2 é o número de elementos a tirar.

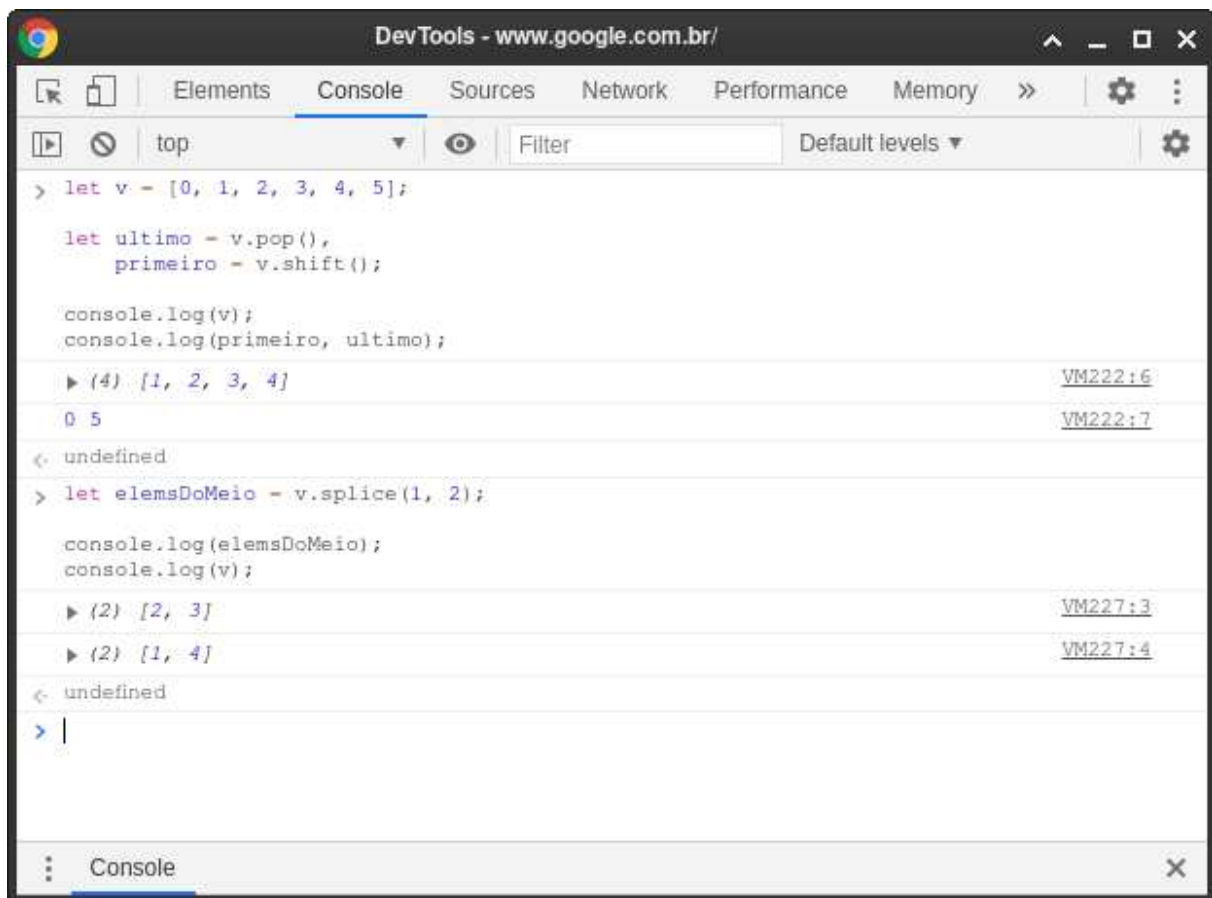


Fig. 8.3: removendo itens de vetor

Criando um novo vetor a partir de um existente

Às vezes, podemos querer criar um novo vetor a partir de um já existente, fazendo algum tipo de alteração nos elementos do atual. Por exemplo, se temos uma lista de números e queremos uma lista do quadrado desses números, podemos criar um novo vetor por meio de um loop:

```
let numeros = [2, 3, 6];
let quadrados = [];
```



```
for (let i = 0; i < numeros.length; i++) {  
  quadrados.push(numeros[i] * numeros[i]);  
}
```

Ou podemos usar map:

```
function quadrado(x) {  
  return x * x;  
}  
  
let numeros = [2, 3, 6];  
let quadrados = numeros.map(quadrado);  
console.log(quadrados);
```

map aceita uma função como primeiro argumento. A função pode ter um, dois ou três parâmetros, sendo eles: (item, índice, vetor), onde item é o item atual; índice, o índice atual; e vetor, o vetor do qual map foi chamado.

```
function valorParaObjeto(elemento, indice) {  
  return {valor: elemento, indice: indice};  
}  
  
let valores = ['a', 2, true];  
let valoresEIndices = valores.map(valorParaObjeto);  
console.log(valoresEIndices);
```



Fig. 8.4: demonstrando map

Filtrando elementos de um vetor

O método `map` normalmente vem acompanhado de outro método igualmente útil chamado `filter`. Como o nome sugere, ele serve para filtrar elementos de um vetor e funciona de forma parecida: seu parâmetro espera uma função predicado que pode ter até 3 parâmetros: (`item`, `indice`, `vetorDeOrigem`). A função é categorizada como “predicado”, pois deve retornar `true` ou `false`. Os itens para os quais a função retornar `false` são descartados do novo vetor.

```

function ehPar(numero) {
  return numero % 2 === 0;
}

let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let pares = numeros.filter(ehPar);

console.log(pares);

```

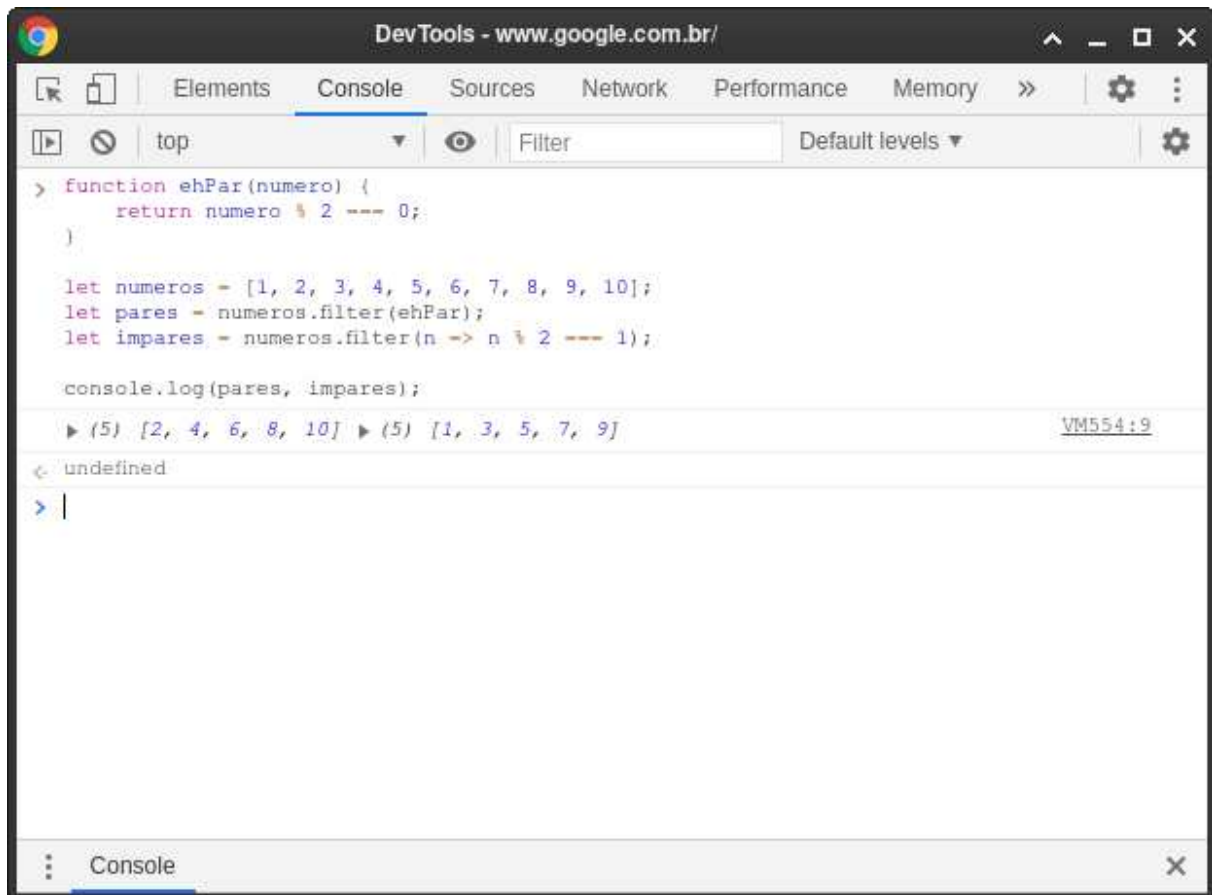


Fig. 8.5: demonstrando filter

Outras operações com vetores

- `vetor.reverse()` inverte a posição dos elementos de um vetor.
- `vetor.every(predicado)` verifica se todos os elementos atendem a uma certa condição.
- `vetor.some(predicado)` verifica se pelo menos um elemento atende a uma certa condição.
- `vetor.find(predicado)` devolve o primeiro elemento que atende à condição ou `undefined` se não encontrar nada.
- `vetor.findIndex(predicado)` devolve o índice do primeiro elemento que atende à condição ou `-1` se não encontrar nada.
- `vetor.includes(valor)` retorna `true`, se valor existe em vetor, ou `false`, caso contrário.
- `vetor.indexOf(valor)` retorna o índice de valor em vetor ou `-1` se valor não for encontrado.

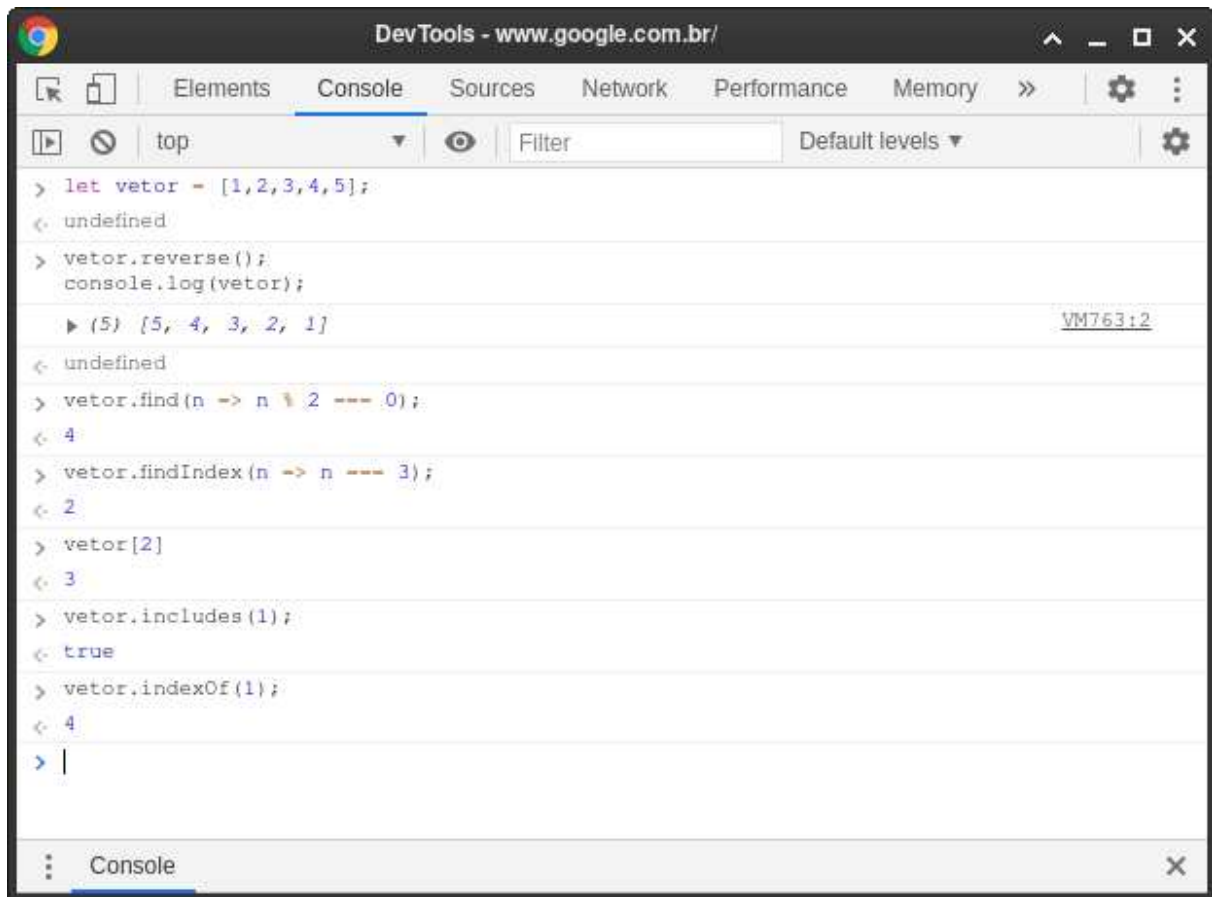


Fig. 8.6: operações de vetores

Revisitando repetições

Na aula 6, aprendemos sobre as estruturas de repetição, ou loops. Aprendemos que um loop `for` é composto por três partes, sendo elas: a inicialização, a condição e a expressão executada no fim de cada iteração. O JavaScript nos permite escrever o `for` de outras duas maneiras além de possibilitar ainda outra forma de iteração em vetores.

Se quisermos que as repetições sejam feitas com base no índice de cada elemento, podemos utilizar `for ... in`:

```

for (let index in vogais) {
  console.log("índice " + index + ": " + vogais[index]);
}

```

Se a preferência for trabalhar diretamente com os elementos, podemos utilizar `for ... of`:

```
for (let vogal of vogais) {
  console.log("Vogal: " + vogal)
}
```

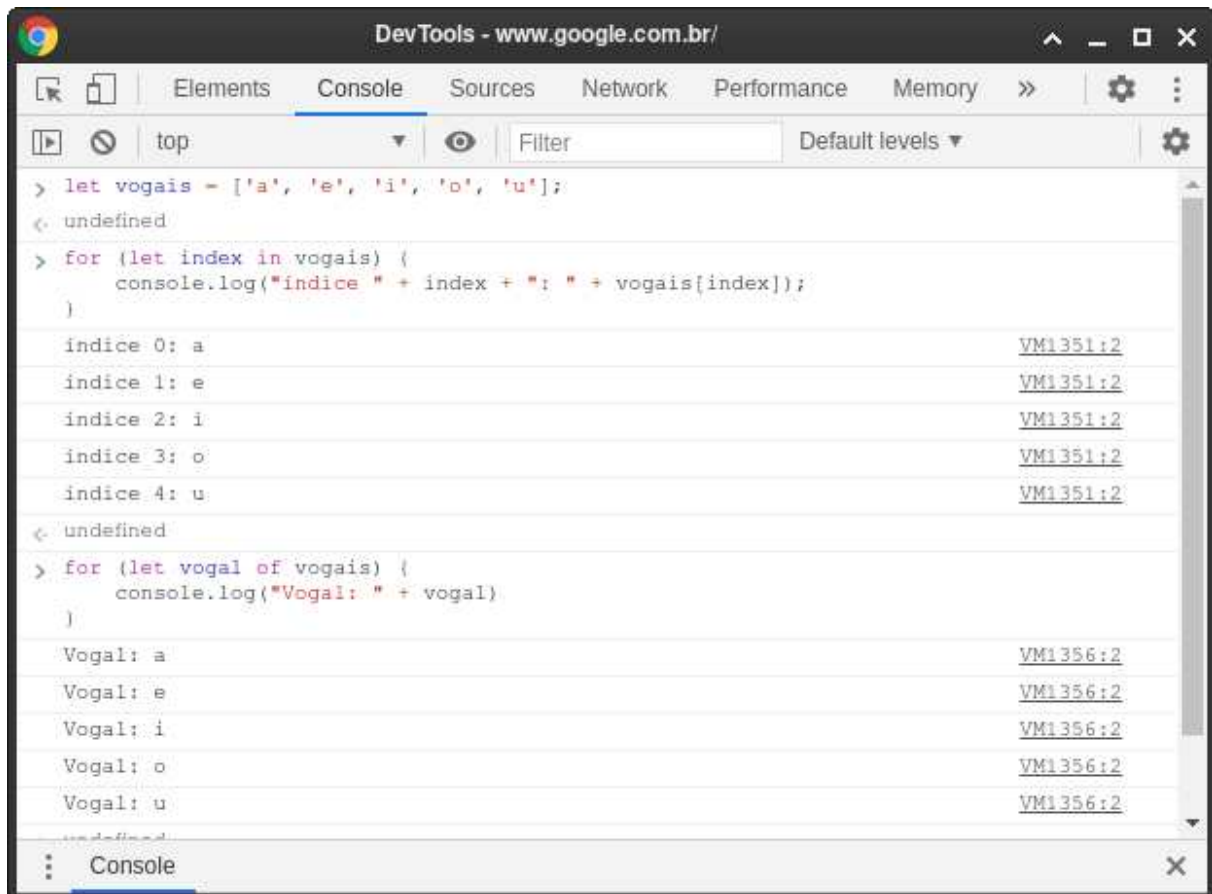


Fig. 8.7: for-in e for-of

Exercícios

1. Escreva uma função que receba uma string como argumento e a retorne sem vogais. Use métodos de vetores para atingir o objetivo. A conversão string -> vetor -> string já está feita para você.
Exemplos:

Entrada	Saída
Hello world	Hll wrld
Chega de pandemia	Chg d pndm

```
function removeVogais(aString) {
  let vetor = Array.from(aString);

  return vetor.join('');
}
```

2. Escreva uma função que valida se dois vetores são circularmente idênticos. Vetores circularmente idênticos são vetores que possuem os mesmos elementos na mesma ordem, mas o primeiro elemento de um vetor pode aparecer em outro índice no outro vetor.

Exemplos:

Entrada	Saída
[1, 2, 3, 4, 5], [3, 4, 5, 1, 2]	true
[1, 2, 3, 4, 5], [1, 2, 3, 5, 4]	false

```
function validaIgualdadeCircular(v1, v2) {
  return false;
}
```

9

Projeto: Casa de Câmbio

Simulando de forma simples uma casa de câmbio com JavaScript

Para o nosso projeto, simularemos uma casa de câmbio simples: escreveremos um programa que compra e vende moedas estrangeiras com uma pequena taxa sobre os valores, que é como eles lucram. Teremos algumas informações sobre as moedas e alguns métodos para a casa lidar com compra, venda e troca de moedas.

As moedas

As moedas têm uma estrutura simples: cada uma tem um nome, uma sigla e um valor base. Este valor é relativo ao real brasileiro. Usaremos um objeto para guardar informações sobre variadas moedas. A estrutura básica deve ser, por exemplo:

Atributo	Tipo de dado	Exemplo
nome	String (texto)	"Dólar"
sigla	String	"USD"
valor	Number (numérico)	5.57

E, por si só, as moedas não possuem nenhum comportamento. Então podemos finalizar por aqui.

A casa de câmbio

Casas de câmbio são um tipo de comércio e, por isso, não podem simplesmente comprar e vender as moedas pelo valor exato delas. Quando vendem a moeda, vendem por um preço mais alto; quando compram, compram por um preço mais baixo.

No nosso sistema, teremos um valor de porcentagem que chamaremos de *taxa*. Essa taxa é o que aplicaremos em cima do valor da moeda para calcular quanto haverá de lucro em cada transação. Podemos dizer, por exemplo, que a casa coloca uma taxa de 10% em cima de cada transação: a moeda estrangeira passa a custar 10% mais cara quando o cliente quer comprar, e 10% mais barata quando o cliente quer vender. Ou seja:

Cliente deseja...	Valor da moeda	Taxa	Preço ajustado
comprar	3.00	10%	3.30
vender	3.00	10%	2.70

Além disso, para “trocar” uma moeda estrangeira por outra, o procedimento é vender a primeira moeda estrangeira para depois comprar a segunda. A estrutura da nossa casa de câmbio poderia ser:

Atributo	Tipo de dado	Exemplo
taxa	Number (numérico, porcentagem)	0.10

Método	Parâmetros	Retorno
proporVenda	moeda, quantidade	preço da moeda com valor ajustado de acordo com a taxa de venda e a quantidade
proporCompra	moeda, quantidade	preço da moeda com valor ajustado de acordo com a taxa de compra e a quantidade
proporTroca	moedaEntrada, qtdEntrada, moedaSaida, qtdSaida	preço da transação de acordo com proporCompra e proporVenda
criarTabela	moedas	um vetor de objetos com as informações de compra e venda de cada moeda

Moedas e seus valores

Para a execução do projeto, usaremos moedas e valores reais do xe.com. Os valores no dia 8 de abril de 2021 são:

Moeda	Sigla	Valor em reais
Dólar	USD	5.56810
Euro	EUR	6.63457
Libra	GBP	7.64738
Iene	JPY	0.05093
Peso	ARS	0.06033

Implementação

Podemos usar objetos do JavaScript para ambas as estruturas. Podemos definir o objeto das moedas da seguinte maneira, junto a uma função auxiliar:

```
function criaMoeda(nome, sigla, valor) {  
  return {nome: nome, sigla: sigla, valor: valor};  
}  
  
let moedas = {  
  usd: criaMoeda('Dólar', 'USD', 5.56810),  
  eur: criaMoeda('Euro', 'EUR', 6.63457),  
  gbp: criaMoeda('Libra', 'GBP', 7.64738),  
  jpy: criaMoeda('Iene', 'JPY', 0.05093),  
  ars: criaMoeda('Peso', 'ARS', 0.06033)  
};
```

Também criamos um objeto para representar a nossa casa de câmbio. Podemos definir 10% como a taxa da casa.

```
let casa = {  
  taxa: 0.1  
}
```

Em seguida, os métodos para calcular e propor os valores de compra e venda. Nelas precisamos calcular o valor ajustado da moeda utilizando a taxa da casa. Quando a casa for vender, adicionamos a porcentagem da taxa ao valor da moeda. Para comprar, subtraímos a porcentagem. O resultado final é a quantidade que o cliente quer multiplicado pelo valor ajustado.

```

casa.proporVenda = function (moeda, quantidade) {
  let valorAjustado = moeda.valor * (1 + this.taxa);
  return quantidade * valorAjustado;
}

casa.proporCompra = function (moeda, quantidade) {
  let valorAjustado = moeda.valor * (1 - this.taxa);
  return quantidade * valorAjustado;
}

```

* this refere-se ao objeto casa. Ou seja, this.taxa é casa.taxa.

O método para propor uma troca faz uso dos dois métodos acima. Recebemos duas moedas e duas quantidades e realizamos os cálculos baseados neles. A proposta final é a diferença entre o valor de venda e de compra.

```

casa.proporTroca = function (moeda1, qtd1, moeda2, qtd2) {
  let propostaCompra = this.proporCompra(moeda1, qtd1),
      propostaVenda = this.proporVenda(moeda2, qtd2);

  return propostaVenda - propostaCompra;
}

```

Por último, o método para criar uma tabela que pode ser exibida com `console.table`. Este método recebe como argumento o objeto de moedas e retorna um vetor de objetos. Cada objeto é uma linha da tabela, e cada atributo do objeto é uma coluna.

```

casa.criarTabela = function (moedas) {
  let tabela = [];

  for (let moeda in moedas) {
    let linha = {};

    linha['Moeda'] = moedas[moeda].nome + ' (' + moedas[moeda].sigla + ')';
    linha['Valor de compra'] = this.proporCompra(moedas[moeda], 1);
    linha['Valor de venda'] = this.proporVenda(moedas[moeda], 1);

    tabela.push(linha);
  }

  return tabela;
}

```

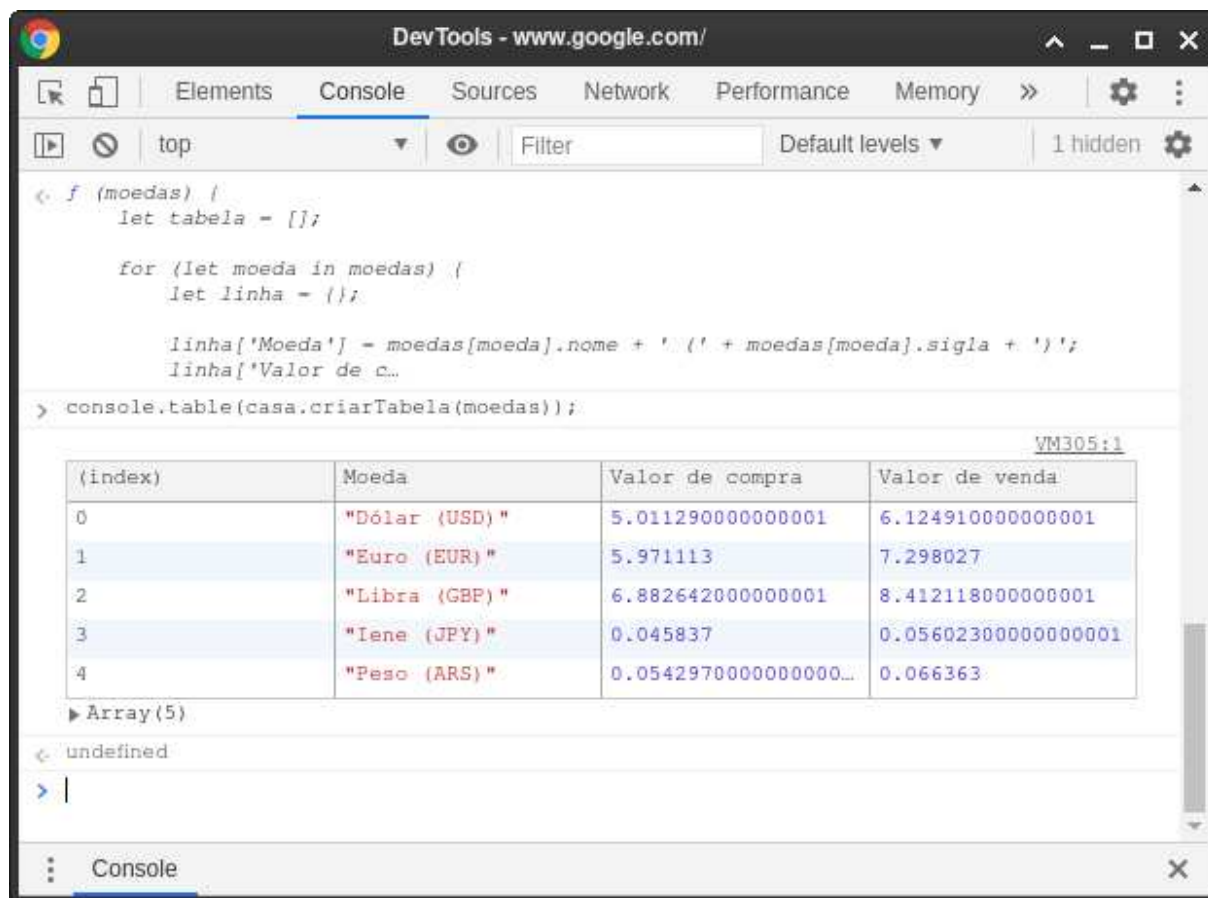


Fig. 9.1: tabela de preços

10

Desafio: Jogo de Damas em JavaScript

O desafio é criar um jogo de Damas em JavaScript. Não é necessário fazer uma interface gráfica ou qualquer coisa muito aprimorada; matrizes e strings são mais do que suficiente.

A sugestão é usar uma letra minúscula para peças comuns e a mesma letra maiúscula para peças dama. Supondo que as peças sejam pretas e brancas, as strings seriam:

Tipo de peça	Cor	Representação
Comum	Preta	p
Dama	Preta	P
Comum	Branca	b
Dama	Branca	B

Jogadores

A princípio, o jogo pode ser feito para dois jogadores para que não seja necessário pensar na lógica de jogadas do computador. No entanto, implementar a lógica do computador é encorajada após a implementação da primeira fase.

Regras

- Peças podem mover-se apenas diagonalmente.
- Peças comuns podem mover-se apenas para frente e uma casa por vez.
- Peças dama podem mover-se para frente ou para trás e por quantas casas for possível.
- Peças adversárias podem ser capturadas se houver um casa livre atrás ou à frente dela.

Por exemplo:

1. Existe uma peça branca b. Não há nenhuma peça atrás de b, apenas uma peça p à sua frente:

```
_ # _
# b #
_ # p
```

1. A peça p, durante seu turno, pode capturar b e finalizar a rodada no espaço que estava vazio atrás de b:

```
p # _
# . #
_ # .
```

1. Porém p não pode capturar b e pular para a diagonal direita em um movimento triangular. A captura precisa ser em linha reta. O movimento seguinte é inválido:

```
_ # p
# . #
_ # .
```

* # denota posições inválidas no tabuleiro; _ denota casas vazias e . denota o caminho da peça.

- Peças podem realizar capturas em cadeia. Exemplos:

1.

```
_ # _ # _ => _ # _ # _
# _ # _ # => # _ # b #
_ # p # _ => _ # . # _
# _ # _ # => # . # _ #
_ # p # _ => _ # . # _
# _ # b # => # _ # . #
```

2.

```
b # _ # _ => . # _ # _
# p # _ # => # . # _ #
_ # _ # _ => _ # . # _
# _ # p # => # _ # . #
_ # _ # _ => _ # _ # b
# _ # _ # => # _ # _ #
```

- Peças comuns que atingem o outro lado do tabuleiro tornam-se damas.

– p P
– b B

- Peças dama podem mover-se para trás e para frente.
- Peças dama podem avançar mais de uma casa em um movimento.
- Peças dama também podem capturar em cadeia, mas *apenas* se houver uma casa livre logo após cada peça alvo.

```
# _ # _ # _ # _ => # _ # _ # _ # _
_ # _ # _ # _ => _ # B # _ # _ # _
# _ # p # _ # _ => # _ # . # _ # _
_ # _ # _ # _ => _ # _ # . # _ # _
# _ # _ # _ # _ => # _ # . # _ # _
_ # p # _ # _ # _ => _ # . # _ # _ # _
# _ # _ # _ # _ => # . # _ # _ # _
B # _ # _ # _ # _ => . # _ # _ # _ # _
```

Objetivos

Você pode seguir o roteiro abaixo para completar o desafio. Sinta-se à vontade para adicionar itens, caso ache interessante.

- ☐ Movimento diagonal simples de peças comuns
 - ☐ Validação de movimento adiante
 - ☐ Validação de casa vazia (não #)
 - ☐ Validação de tabuleiro (a peça não deve sair do tabuleiro)
- ☐ Captura de peças por peças comuns
 - ☐ Captura simples
 - ☐ Captura em cadeia (opcional)
 - ☐ Validação de casa vazia atrás da peça alvo
- ☐ Aprimoração de peça comum para peça dama no outro lado do tabuleiro
- ☐ Movimento diagonal múltiplo de peças dama
 - ☐ Validação de movimento adiante
 - ☐ Validação de movimento para trás
 - ☐ Validação de tabuleiro
- ☐ Captura de peças por peças dama
 - ☐ Captura simples
 - ☐ Captura em cadeia (opcional)
 - ☐ Validação de casa vazia atrás da peça alvo
- ☐ Jogadores
 - ☐ Humano x Humano
 - ☐ Humano x Máquina (opcional)
- ☐ Validação de fim de jogo quando um dos jogadores perder todas as peças

Implementação

A implementação deverá ser toda em JavaScript. Abaixo, você vê uma sugestão de estrutura básica, que você pode seguir se desejar. O tabuleiro foi montado pensando na visualização com `console.table`.

```
let jogo = {
  pecas: {
    brancas: 12,
    pretas: 12
  },
  tabuleiro: [
    ['p', '#', 'p', '#', 'p', '#', 'p', '#'],
    ['#', 'p', '#', 'p', '#', 'p', '#', 'p'],
    ['p', '#', 'p', '#', 'p', '#', 'p', '#'],
    ['#', ' ', '#', ' ', '#', ' ', '#', ' '],
    [' ', '#', ' ', '#', ' ', '#', ' ', '#'],
    ['#', 'b', '#', 'b', '#', 'b', '#', 'b'],
    ['b', '#', 'b', '#', 'b', '#', 'b', '#'],
    ['#', 'b', '#', 'b', '#', 'b', '#', 'b']
  ]
};
```

As funções e métodos ficam por sua conta. Você decide quantas funções ou métodos o jogo terá.

Boa sorte!