- Programas funcionam manipulando valores, como o número `3.14` ou o texto `Juliana e Leonardo`. Os tipos de valores que podem ser representados e manipulados em uma linguagem de programação são conhecidos como tipos, e uma das características mais fundamentais de uma linguagem de programação é o conjunto de tipos que ela suporta.

Variáveis

- Quando um programa precisa reter um valor para uso futuro, ele atribui o valor a (ou "armazena" o dado em) uma variável. As variáveis têm **nomes** e permitem o uso desses nomes em nossos programas para se referir a valores. A maneira como as variáveis funcionam é outra característica fundamental de qualquer linguagem de programação.
- Os tipos de JavaScript podem ser divididos em duas categorias: tipos **primitivos** e tipos de **objetos**. Os tipos primitivos do JavaScript incluem números, palavras ou texto (conhecidas como **strings**) e valores booleanos (conhecidos como **booleanos**).

Tipos numéricos

- Como vimos em aula, podemos armazenar números de diferentes formas:
- `const idade = 28`
- `const pi = 3.14`
- **COPIAR CÓDIGO**
- Dica: podemos utilizar o número PI através com o código `Math.PI`.
- O ponto flutuante pode ter um ponto decimal; eles usam a sintaxe tradicional para números reais. Um valor real é representado como a parte integral do número, seguido por um ponto decimal e a parte fracionária do número.
- Pontos flutuantes também podem ser representados usando notação exponencial: um número real seguido pela letra e (ou E), seguido por um sinal opcional de mais (+) ou menos (-), e por um expoente inteiro. Essa notação representa o número real multiplicado por 10 à potência do expoente.
- Divisão por zero não é um erro em JavaScript: ele simplesmente retorna "*Infinity*". No entanto, há uma exceção: zero dividido por zero não tem um valor bem definido e o resultado dessa operação é o valor especial não numérico NaN.
- `var a = 10`
- `var b = 0`
- `console.log(a/b) // Infinity`
- **COPIAR CÓDIGO**
- `var a = 0`
- `var b = 0`
- `console.log(a/b) // NaN`
- **COPIAR CÓDIGO**
- Acabamos de ver que usamos o tipo `string` sempre que queremos trabalhar com dados de texto. Mas se pararmos

para pensar, vários idiomas utilizam caracteres diferentes, como acentos e ideogramas. Como as linguagens de programação lidam com isso? E o que dizer dos *emojis* :question:? Você já visitou algum site e notou que os caracteres dos textos não pareciam corretos, que no lugar de alguns deles aparecia um sinal de interrogação, quadrados ou traços?

- Isso tudo tem a ver com a **codificação de caracteres**, ou *character encoding*. Nas últimas décadas, foram desenvolvidos diversos conjuntos de caracteres especiais, cada um com seus próprios códigos, para que pessoas que escrevem e leem em linguagens diferentes do inglês pudessem utilizar computadores com seus próprios idiomas. E como isso funciona?
- Para que o computador consiga **decifrar** um caractere especial, é preciso utilizar um sistema específico que tenha basicamente um código para cada caractere, e que o computador possa acessá-lo para fazer a conversão - uma ideia similar a que está por trás da criptografia.
- Foram desenvolvidos diversos conjuntos de caracteres, desde os específicos de cada linguagem como Western, Latin-US, Japanese e assim por diante, até o ASCII (*American Standard Code for Information Interchange* ou ”Código Padrão Americano para o Intercâmbio de Informação”). e a partir de 2007 foi adotado o formato Unicode. O padrão UTF (de *Unicode Transformation Format* ou “formato de conversão de unicode”, em tradução livre) é utilizado como padrão na web até hoje.
- O Unicode tem códigos específicos para “cifrar” e “decifrar” caracteres de mais de 150 idiomas antigos e modernos, e também diversos outros conjuntos de caracteres como símbolos matemáticos e inclusive *emojs*. A Wikipedia tem uma lista extensa de todas as tabelas com os códigos Unicode e os caracteres, como por exemplo os que estão abaixo:

caractere	**UTF-16**	**descrição oficial**
$	U+0024	DOLLAR SIGN
A	U+0041	LATIN CAPITAL LETTER A
✅	U+2705	CHECK MARK
あ	U+3040	HIRAGANA LETTER SMALL A

- Podemos testar a transformação/conversão do código Unicode em caractere utilizando o `console.log()`. Faça o teste:

```
const cifrao = '\u0024'
const aMaiusculo = '\u0041'
const tique = '\u2705'
const hiragana = '\u3041'

console.log(cifrao)
console.log(aMaiusculo)
console.log(tique)
console.log(hiragana)
```

COPIAR CÓDIGO

- Os caracteres `\u` no início do código são **caracteres de escape** que usamos para sinalizar ao JavaScript de que estamos falando de códigos Unicode, e não de strings de texto usuais.
- O JavaScript usa, por padrão, o UTF-16. O número 16 está relacionado aos espaços em bits ocupados por cada caractere, 16 neste caso. Não vamos nos aprofundar na relação entre tipos de dados e espaço de memória ocupado por cada tipo - você pode pesquisar mais sobre o assunto, assim como sobre o que são caracteres de escape! - mas por enquanto é bacana vermos na prática como o Unicode funciona.
- Bancos de dados podem aceitar outros tipos de codificação de caracteres, o que faz sentido se pensarmos que o UTF-16 utiliza uma quantidade relativamente grande de espaço em memória para salvar cada caractere. 16 bits parece pouco, mas algumas vezes os bancos precisam salvar quantidades enormes de dados! Porém, com as tecnologias de armazenamento e tráfego de dados que temos hoje, esta já não é uma preocupação tão grande, a não ser em casos muito específicos. Já não é muito comum utilizar uma codificação diferente da UTF em bancos mesmo em caso de grandes volumes de dados, mas sempre vai depender **muito** do caso.
- O JavaScript traz em sua biblioteca-base vários métodos que usamos para manipular strings de texto: alterar de maiúsculas para minúsculas, contar quantas letras tem uma palavra, retirar espaços, juntar duas strings, etc.
- Vamos pensar em alguns exemplos práticos para fazer esse tipo de alteração. Por exemplo, para padronizar uma comparação entre strings:

```
const cidade = "belo horizonte";
const input = "Belo Horizonte";

console.log(cidade === input); // false
```

COPIAR CÓDIGO

- Nós, como pessoas, conseguimos perceber o valor das variáveis `cidade` e `input` como sendo da mesma cidade, Belo Horizonte. Porém, para o JavaScript, ambos os dados são apenas sequências de caracteres, e a comparação vai falhar, pois como já vimos, **o JavaScript diferencia minúsculas e maiúsculas, tanto nos valores dos dados quanto no código que escrevemos**.
- Uma das formas de tratar isso é padronizando todos os inputs para o formato de texto que será comparado antes mesmo de fazer a comparação. Nesse caso, transformando todos os caracteres em letras minúsculas.
- `const cidade = "belo horizonte";`
- `const input = "Belo Horizonte";`
-
- `const inputMinusculo = input.toLowerCase();`
-
- `console.log(cidade === inputMinusculo); // true`
- **COPIAR CÓDIGO**
- Acima, vemos um dos **métodos de string nativos do JavaScript** em ação, o `toLowerCase()` que converte todos os caracteres da string informada (no caso, `input`) para letras minúsculas (se forem algarismos, nada é convertido). Você pode conferir mais sobre este método no MDN.
- Outro exemplo: qualquer inserção de texto que exija uma quantidade mínima de caracteres, como uma senha ou um nome. A propriedade `length` pode ser utilizada para sabermos quantos caracteres uma string contém:
- `const senha = "minhaSenha123"`
- `console.log(senha.length) // 13 caracteres`
- **COPIAR CÓDIGO**
- A propriedade `length` é muito usada no dia a dia do desenvolvimento web. Você pode descobrir mais sobre ela aqui.
- Percebeu que `length` não leva parênteses no final da palavra? Há uma diferença entre **métodos** e **propriedades** que não vamos abordar durante este curso, mas vamos deixar aqui a dica caso tenha curiosidade! ;)
- **STRING**
- PODE SE USAR UMA ASPA OU DUAS E É USADO PARA GUARDAR CARACTERES
- e pode fazer cocatenação
- ex: const citacao = “meu nome é”;
- const meuNome = “Leonardo”;
- console.log(citacao + meuNome);

O JavaScript traz em sua biblioteca-base vários métodos que usamos para manipular strings de texto: alterar de maiúsculas para minúsculas, contar quantas letras tem uma palavra, retirar espaços, juntar duas strings, etc.

Vamos pensar em alguns exemplos práticos para fazer esse tipo de alteração. Por exemplo, para padronizar uma comparação entre strings:

```
const cidade = "belo horizonte";
const input = "Belo Horizonte";


console.log(cidade === input); // false
```

COPIAR CÓDIGO

Nós, como pessoas, conseguimos perceber o valor das variáveis `cidade` e `input` como sendo da mesma cidade, Belo Horizonte. Porém, para o JavaScript, ambos os dados são apenas sequências de caracteres, e a comparação vai falhar, pois como já vimos, **o JavaScript diferencia minúsculas e maiúsculas, tanto nos valores dos dados quanto no código que escrevemos**.

Uma das formas de tratar isso é padronizando todos os inputs para o formato de texto que será comparado antes mesmo de fazer a comparação. Nesse caso, transformando todos os caracteres em letras minúsculas.

```
const cidade = "belo horizonte";
const input = "Belo Horizonte";


const inputMinusculo = input.toLowerCase();

console.log(cidade === inputMinusculo); // true
```

COPIAR CÓDIGO

Acima, vemos um dos **métodos de string nativos do JavaScript** em ação, o `toLowerCase()` que converte todos os caracteres da string informada (no caso, `input`) para letras minúsculas (se forem algarismos, nada é convertido). Você pode conferir mais sobre este método no MDN.

Outro exemplo: qualquer inserção de texto que exija uma quantidade mínima de caracteres, como uma senha ou um nome. A propriedade `length` pode ser utilizada para sabermos quantos caracteres uma string contém:

```
const senha = "minhaSenha123"

console.log(senha.length) // 13 caracteres
```

COPIAR CÓDIGO

A propriedade `length` é muito usada no dia a dia do desenvolvimento web. Você pode descobrir mais sobre ela aqui.

Percebeu que `length` não leva parênteses no final da palavra? Há uma diferença entre **métodos** e **propriedades** que não vamos abordar durante este curso, mas vamos deixar aqui a dica caso tenha curiosidade! ;)

Você pode conferir a lista completa de métodos de string do MDN (são vários), com a descrição de cada um, e praticar com os exemplos.

- **VAR,LET E CONST**
- **são formas de declarar as variáveis**
- var é global consegue declarar ela em qualquer lugar do código não é preiso declarar antes
- let ja serve mas para aquele bloco que ele foi criado é preciso declarar antes
- const tem o mesmo escopo do let so que não se pode reatribuir um valor a const que já foi declarada,pórem ela pode ser modificada
- var já não é tão usada

Um detalhe muito importante, mas que às vezes não percebemos quando começamos a programar, é que **cada linguagem possui seus próprios padrões**. Eles servem não somente para a escrita de códigos que funcionem, mas também para criar nomes de variáveis, estruturar um programa e muito mais.

A primeira coisa que precisamos ter em mente é que **o JavaScript é *case-sensitive*, ou seja, diferencia maiúsculas e minúsculas**. Isso significa que tudo o que escrevemos, sejam instruções próprias da linguagem (como `console.log`) ou quando damos nome a uma variável, tem que ser feito em um mesmo padrão, o que inclui a questão de maiúsculas e minúsculas.

Para ilustrar, o JavaScript trata os quatro exemplos abaixo como variáveis diferentes e não apresentará nenhum erro se você executar o programa:

- `const minhaVar = 1;`
- `const MinhaVar = "texto";`
- `const minhavar = "3";`
- `const MINHAVAR = 2;`
-

```
console.log(minhaVar,
  MinhaVar, minhavar,
  MINHAVAR)
```

COPIAR CÓDIGO

Podemos ver que, em um programa muito grande, a possibilidade de problemas é grande. Então como sabemos a forma certa de nomear? Aí entra o que chamamos de convenções, para padronizar estes aspectos do código.

Existem várias convenções para nomes e cada linguagem de programação tem o seu. Seguem alguns deles:

- `camelCase`: Inicia com letra minúscula e a primeira letra de cada palavra em seguida é escrita com letra maiúscula. Por exemplo: `minhaVar` ou

`senhaDoUsuario`. **Esta é a convenção utilizada pelo JavaScript para variáveis e funções.**

- `snake_case`: Os espaços são substituídos pelo caractere _ (underline), com todas as palavras em letra minúscula. Por exemplo: `minha_variavel` ou `senha_do_usuario`.
- `kebab-case`: Similar ao anterior, porém com os espaços substituídos por hífens. Por exemplo: `minha-var` ou `senha-do-usuario`.
- `PascalCase`: Similar ao `CamelCase`, porém neste caso todas as palavras começam com letra maiúscula. Por exemplo: `MinhaVar` ou `SenhaDoCliente`.

Importante: Nunca utilize espaço nem caracteres especiais, nem inicie os nomes das variáveis com números.

Quando falamos de convenção, estamos falando de boas práticas e padronização. Se você utilizar qualquer um dos padrões acima para nomear variáveis com JavaScript, seu código continuará funcionando, mas seguir as convenções é parte de desenvolver um código legível e bem escrito.

Esse é um assunto vasto e com muitos detalhes, e é parte do nosso trabalho no cotidiano como pessoas que desenvolvem garantir que os chamados **guias de estilo** definidos para um produto de código sejam seguidos.

Você pode ir aprendendo os detalhes aos poucos, enquanto estuda, e observá-los sendo aplicados nos códigos que vê por aí.

- **TIPO BOOLEAN**
- TRUE-VERDADEIR
- FALSE-FALSO
- === faz a comparação com os valores
- **TRUTHY E FALSY**
- 0 => false
- 1=> true
- null-representa vazio ou nada,só que le foi criado como object(um bug)
- undefined - coloca variável mas nao define ela
- typeof palavra chave que pergunta ao javascript qual é o tipo de dado guardado na variável
- **CONVERSÕES DE TIPOS**
- conversão implícita- converter um dado em outro ex: string em number pode trazer alguns bugs
- conversão explícita-
  Number() transformar string em número
  String() number em string
  console.log(numero+number(numeroString);

se no tipo numero tiver algum caracter vai sair :Nan(not a number)

String()

Vamos fazer alguns exemplos de conversão de números e booleanos através de `String()`:

```
let telefone = 12341234;
```

```
console.log("O telefone é " + String(telefone)); // teremos a conversão do número 12341234 para uma string "12341234" e assim poderemos fazer a concatenação entre as strings
```

COPIAR CÓDIGO

Outra opção para transformarmos um valor em String é usar o toString():

```
let telefone = 12341234;

console.log("O telefone é " + telefone.toString(); // o .toString() é uma outra forma para  fazer essa conversão, que é mais parecida com outras linguagens de programação.
```

COPIAR CÓDIGO

```
let usuarioConectado= false;
console.log (String( usuarioConectado ) ); // teremos a conversão da booleana para string, nesse caso teremos uma string "false".
usuarioConectado= true;
console.log (String( usuarioConectado ) ); // agora teremos uma string "true".
```

COPIAR CÓDIGO

Number()

Vamos fazer alguns exemplos de conversão de textos e booleanos através de Number():

```
// Vamos calcular a área de um retângulo
let largura = "10";
let altura = "5";

console.log( Number(largura) * Number(altura)); // teremos a conversão de String para números, possibilitando a realização da da multiplicação
```

COPIAR CÓDIGO

Podemos usar o operador de soma + para fazer a conversão de textos para números, colocando-os antes das variáveis:

```
let largura = "10";
let altura = "5";
```

```
console.log( + largura * +
altura); // teremos a
conversão de String para
números realizado usando o +
antes das variáveis
```

COPIAR CÓDIGO

```
let meuNome = "leonardo";
console.log(Number(meuNome));
// como a variável meuNome
não contém apenas números ele
retorna o erro NaN (Not a
Number, não é número);

console.log( + meuNome); // a
conversão também retornará
NaN
```

COPIAR CÓDIGO

```
let usuarioConectado= false;
console.log (Number(
usuarioConectado ) ); //
teremos a conversão da
booleana para número, sendo
que false (falso) retorna o
número 0.
usuarioConectado= true;

console.log (Number(
usuarioConectado ) ); //
agora teremos a conversão de
true (verdadeiro) para o
número 1.
```

COPIAR CÓDIGO

Dica de boas práticas: Apesar do JavaScript fazer a maioria das conversões de forma correta, problemas podem aparecer, então é sempre bom fazer as conversões de forma explícita. Não é comum usar o operador de soma para fazer a conversão para números, mas este uso é possível. Conversões de booleanos não costumam ser muito usados, mas são possíveis.

- Já aprendemos a declarar variáveis, sejam elas let ou const, utilizando a palavra-chave e um nome que escolhemos para a variável. Chamamos este nome justamente de **identificador**, e o ideal é que sejam sempre o mais explicativos possível:
- `let cpfUsuario = "12312312312"`
- COPIAR CÓDIGO
- Mas o que acontece se tentarmos identificar uma variável com um termo que faça parte da linguagem, como nos casos abaixo?
- `let var = 0;`
- `let if = 0;`
- `let const = "Alura";`
- COPIAR CÓDIGO

- Faça o teste para ver que o JavaScript não consegue reconhecer estas palavras-chave como identificadores e nem interpretar o que deve ser executado nestas linhas. Isso acontece porque `var`, `if` e `const` são **palavras reservadas** do JavaScript. Ou seja, não podemos usá-las para dar nomes (identificar) variáveis, funções ou outros blocos de código que precisem de identificadores.
- Por outro lado, os exemplos abaixo são aceitáveis:
- `let varInicial = 0;`
- `let ifFalse = 0;`
- `let constDeTexto = "Alura";`
- `COPIAR CÓDIGO`
- No JavaScript, algumas palavras são totalmente reservadas (não podem ser utilizadas como identificador em nenhum caso), enquanto outras podem ser utilizadas dependendo do contexto, e ainda outras não podem ser consideradas totalmente reservadas por razões de compatibilidade com versões mais antigas da linguagem, como é o caso de `let` (lembrando que, até o ES6, só era possível declarar variáveis com `var` e `let`, que vem do verbo em inglês "permitir").
- A melhor prática, nesse caso, é não utilizar nenhum dos termos da lista abaixo como identificadores, seja de variáveis, funções, classes ou qualquer outro bloco que precise de um nome. **As únicas exceções são `from`, `set` e `target`, que são seguras e comumente utilizadas.**

- `arguments`
- `as`
- `async`
- `await`
- `break`
- `case`
- `catch`
- `class`
- `const`
- `continue`
- `debugger`
- `default`
- `delete`
- `do`
- `else`
- `eval`
- `export`
- `extends`
- `false`

- `finally`
- `for`
- `from`
- `function`
- `get`
- `if`
- `import`
- `in`
- `instanceof`
- `let`
- `of`
- `new`
- `null`
- `return`
- `set`
- `static`
- `super`
- `switch`
- `target`
- `this`
- `throw`
- `true`
- `try`
- `typeof`
- `var`
- `void`
- `while`
- `with`
- `yield`
- **COPIAR CÓDIGO**
- Como as linguagens estão em constante desenvolvimento, o JavaScript também restringe o uso de mais algumas palavras que podem ser utilizadas nas próximas versões:
- `enum`
- `implements`
- `interface`
- `package`
- `private`
- `protected`
- `public`
- **COPIAR CÓDIGO**
- Dica de boas práticas: sempre procure nomear/identificar seu código da forma mais **semântica** possível, pensando em qual é o dado que está sendo salvo na variável e para que ele será utilizado. Além de evitar palavras reservadas, faz com que o código fique mais compreensível e de leitura mais fluida.
-

- **javaScript e NodeJs**
- js tem a tipagem dinâmica ou seja não é preciso declarar o tipo(number,string0 e poder trocar nas variáveis tipo Untyped, ela é multiparadigma(consegue resolve um problema de várias formas) . egmascript é seu nome real, ela é uma linguagem interpretada(o código é executado instantaneamente não precisa de um compilador) e possui suporte para funcional, orientado a objetos ou lógico por exemplo.
- Node js - é um intrepertador de js geralmente para back end e assim conseguimos rodar js fora do navegador nesse ambiente
- **ERROS E STACKTRACE**

syntaxError;um erro de sintaxe
referenceError: erro de definiçao de variavel

Cada linguagem de programação tem sua própria forma de lidar com erros. O JavaScript começa dividindo cada tipo de erro possível em algumas categorias:

- `RangeError`: Quando o código recebe um dado do tipo certo, porém não dentro do formato aceitável. Por exemplo, um processamento que só pode ser feito com números inteiros maiores ou iguais a zero, mas recebe `-1`.
- `ReferenceError`: Normalmente ocorre quando o código tenta acessar algo que não existe, como uma variável que não foi definida; muitas vezes é causado por erros de digitação ou confusão nos nomes utilizados, mas também pode indicar um erro no programa.
- `SyntaxError`: Na maior parte dos casos ocorre quando há erros no programa e o JavaScript não consegue executá-lo. Os erros podem ser métodos ou propriedades escritos ou utilizados de forma incorreta, por exemplo, operadores ou sinais gráficos com elementos a menos, como esquecer de fechar chaves ou colchetes.
- `TypeError`: Indica que o código esperava receber um dado de um determinado tipo, tal qual uma string de texto, mas recebeu outro, como um número, booleano ou null.

O NodeJS trabalha com outros tipos específicos de erro que não vamos abordar neste momento, mas que você pode sempre consultar na documentação oficial.

Além do tipo de erro, o terminal também vai dar outras informações, como o nome do arquivo e linha onde foi detectado o erro. Muitas vezes isso já basta para identificar e corrigir, mas existem também casos onde o erro não é detectado pelo JavaScript na linha onde o código é declarado, por exemplo, mas onde ele é executado. Por isso é importante praticar sempre a leitura dos erros e da *stacktrace* e nunca pular esta etapa.

- **CONSOLE.API**
- console nod e navegador que serve pra jogar dados,variáveis ou texto pra fora
  .log - cria um registro no console e podemos colocar qualquer informação nesse registro

  console.error() é se fizer algo irar gerar o erro

Embora seja o mais utilizado, `.log()` é um dos vários métodos que podemos utilizar para exibir informações na chamada “saída padrão” - o terminal - enquanto estamos desenvolvendo uma aplicação. A palavra *log* significa algo como “registro”, então

este método apenas registra no terminal o que passamos entre os parênteses, por exemplo o conteúdo de uma variável ou o resultado de uma operação.

Entre os outros métodos, existem:

- `console.error()` para exibir mensagens de erro;
- `console.table()` para visualizar de forma mais organizada informações tabulares;
- `console.time()` e `console.timeEnd()` para temporizar período que uma operação de código leva para ser iniciada e concluída;
- `console.trace()` para exibir a *stacktrace* de todos os pontos (ou seja, os arquivos chamados) por onde o código executado passou durante a execução.

A documentação oficial do NodeJS dá exemplos sobre como utilizar cada um destes métodos e mais outros da lista. É uma documentação bastante extensa, mas não se preocupe! Você não precisa decorar a lista completa, já que ela está sempre disponível para consulta. Para a maior parte dos casos de exemplo, vamos continuar usando `console.log()`.

- **OPERADORES**
- Operadores de Comparação - ==(compara mas antes faz a conversão implicita); ===(compara e não faz a comparação implicita) ele compara valor e o tipo as boas praticas pedem === e fazer conversão explicita
-

Atribuição de adição	x + y
Atribuição de subtração	x - y
Atribuição de multiplicação	x * y
Atribuição de divisão	x / y

- `||`: Operador "ou", retorna `true` caso uma condição seja válida;
- `&&`: Operador "e", retorna `true` somente se todas as condições forem válidas;
- `!=` e `!==`: Operadores "não igual" e "estritamente não igual", utilizados para comparação, da mesma forma que `==` e `===` retornam `true` ou `false`.
- Operador Ternário - (praticamente um if mas em uma linha ou mais reduzida), console.log(idadeCliente >= idadeMinima ? "cerveja" : "suco") primeiro condição depois vem caso verdadeira depois caso falso. se for condições pequenas é legal usar, se não melhor if e else
- Template Literal - template string é outra forma de escrever strings usa ` e usa ${} ex: const apresentacao= `meu nome é ${nome}

é escrito um operador ternário, com o qual fazemos uma comparação entre valores digitando um `?`, seguido da possibilidade *true*, um `:` e a

possibilidade *false*, ou seja, `comparação ? true : false`

-) Operador ternário: Vimos que é possível não apenas exibir o valor de variáveis utilizando o `${}`, mas também fazer operações com JavaScript - por exemplo, condicionais - e inserir o correspondente ao `true` ou `false` na string de texto.
- operador ternário", que se deve ao fato de termos 3 operadores juntos em uma única linha para desempenhar uma tarefa e devolver um resultado.

```
const pedido = `${nome} diz: "por favor, quero beber ${idade >= 18 ? bebidaMaior : bebidaMenor}"`
```

- `console.log(pedido)`
- FUNÇÕES
- usa para usar um pedaço do código no momento que a gente quer
- ex: function imprimeTexto(texto) {
  console.log(texto)

}
imprimeTexto("oi");
imprimeTexto("outro texto");

- 1 Declara a função
- 2 executa a função (1 ou + vezes);
- as Três formas de escrever funções
- o return(ultima linha do código para ser executado) é muito importante para especificar o que quer que volte de resposta ex: function soma(){
  return 2 + 2
  }
  console.log(soma())

JavaScript nos oferece algumas funções prontas, como é o caso de funções matemáticas (Math em inglês), alguns exemplos são:

- `Math.round()`: Faz o arredondamento (round em inglês) de um número de ponto flutuante para o inteiro mais próximo.
  - `Math.round(4.3)` retorna 4
  - `Math.round(3.85)` retorna 4
  - `Math.round(2.5)` retorna 3, quando o número termina com 0.5 a função arredonda para cima
- `Math.ceil()`: Faz o arredondamento para o valor mais alto, também conhecido como teto (ceil em inglês).
  - `Math.ceil(5.2)` retorna 6
- `Math.floor()`: Faz o arredondamento para o valor mais baixo, também conhecido como piso (floor em inglês).
  - `Math.floor(5.2)` retorna 5
- `Math.trunc()` : Desconsidera os números decimais, o que é conhecido como truncamento.
  - `Math.trunc(4.3)` retorna 4
  - `Math.trunc(4.8)` retorna 4
- `Math.pow()` : Faz a exponenciação de 2 números, quando for

simples, como ao quadrado(2) ou cubo(3). Recomenda-se usar a multiplicação por ser mais rápido.
    - `Math.pow(4 , 2)` retorna 4^2 = 16
    - `Math.pow(2.5 , 1.5)` retorna 2.5^(3/2) = 3.9528 ...
- `Math.sqrt()` : Retorna a raiz quadrada de um número.
    - `Math.sqrt(64)` retorna 8
- `Math.min()`: Retorna o menor valor entre os argumentos.
    - `Math.min(0, 150, 30, 20, -8, -200)` retorna -200
- `Math.max()`: Retorna o maior valor entre os argumentos.
    - `Math.max(0, 150, 30, 20, -8, -200)` retorna 150
- `Math.random()`: Retorna um valor randômico (random em inglês) entre 0 e 1, então não teremos um valor esperado para receber.
    - `Math.random()` retorna 0.7456916270759015
    - `Math.random()` retorna 0.15449802102729304
    - `Math.random()` retorna 0.4214269561951203
    -

❖ PARAMETROS
❖ serve para função receber informações para ela executar corretamente
❖ ordem dos parâmetros
❖ os nomes vão ser restritos durante sua execução depois eles ficam livres
❖ boa pratica uma função ter poucos argumentos(pode quebrar se tiver muito
❖ Os parâmetros e o retorno das funções são utilizados de acordo com cada caso específico. Isso significa que nem sempre todas as funções que escrevemos vão precisar de um ou de outro para fazer o que precisam. Abaixo temos mais exemplos para entender melhor algumas situações.
❖ **Função sem retorno e sem parâmetro: A função abaixo apenas executa uma instrução, sem a necessidade de disponibilizar o resultado para o restante do código. Neste exemplo escolhemos usar uma string fixa, então não há necessidade de parâmetros.**
❖ `function cumprimentar(){`
❖ `  console.log('oi gente!')`
❖ `}`
❖
❖ `cumprimentar()`
❖ `COPIAR CÓDIGO`
❖ **Função sem retorno, com parâmetro: similar à anterior, porém agora a função recebe, via parâmetro, o nome da pessoa a ser cumprimentada. Dessa forma é possível reaproveitar a função para**

que funcione de maneira parecida com o nome de qualquer pessoa (desde que esteja no formato de dado `string`.

- ```
  function cumprimentaPessoa(pessoa){
  ```
- ```
    console.log(`oi, ${pessoa}!`)
  ```
- ```
  }
  ```
-
- ```
  cumprimentaPessoa('Helena')
  ```
- **COPIAR CÓDIGO**
- **Função com retorno, sem parâmetro: É possível combinar funções para que cada uma controle apenas uma parte do código e elas trabalhem juntas.**
- No caso abaixo, a função `cumprimentar()` não precisa receber nenhum parâmetro. Mas logo abaixo vemos que ela está sendo utilizada para montar uma string na função `cumprimentaPessoa(nomePessoa)`. **Isso significa que a string "`Oi gente!`" deve estar disponível para outras partes do programa - ou seja, deve ser retornada com o uso da palavra-chave `return`.**
- ```
  function cumprimentar(){
  ```
- ```
    return 'Oi gente!'
  ```
- ```
  }
  ```
-
- ```
  function cumprimentaPessoa(nomePessoa) {
  ```
- ```
    console.log(`${cumprimentar()} Meu nome é ${nomePessoa}`)
  ```
- ```
  }
  ```
-
- ```
  cumprimentaPessoa('Paula') // "Oi gente! Meu nome é Paula"
  ```
- **COPIAR CÓDIGO**
- A função `cumprimentaPessoa(nomePessoa)` recebe como parâmetro uma string onde podemos passar qualquer nome no momento em que executamos (ou chamamos) a função. Quando isso acontecer, a função `cumprimentar()` será executada também, e seu valor de retorno - a string `Oi gente!` - vai ocupar o lugar do `${}` onde a função está sendo chamada.
- **Função com `return` e mais de um parâmetro: Lembrando que as funções podem receber a quantidade de parâmetros necessária, e que o JavaScript identifica os parâmetros pela ordem! Ou seja, no exemplo abaixo o parâmetro `numero1` se refere a `15`, o parâmetro `numero2` se refere a `30` e o parâmetro**
  ```

numero3 se refere a 45. Somos nós, que estamos desenvolvendo o código, que damos os nomes aos parâmetros de acordo com o dado que a função espera receber - no caso, números.

- `function operacaoMatematica(numero1, numero2, numero3) {`
- `  return numero1 + numero2 + numero3`
- `}`
-
- `operacaoMatematica(15, 30, 45) // 90`
- **COPIAR CÓDIGO**
- **Parâmetros x argumentos: Na prática se referem ao mesmo tipo de dado; algumas documentações se referem a *parâmetros* no momento em que a função é definida (no caso, numero1, numero2, etc) e *argumentos* como os dados que utilizamos para executar a função (ou seja, 30, 45, etc).**
- Ainda há muito o que estudar no tema de funções, então pratique bastante pois parâmetros e retorno são conceitos essenciais.
- ARROW FUNCTION
- diminui o código(não precisa de chaves , return)
- ex: const apresentaArrow =nome => 'meu nome é ${nome}';

```
const soma = (num1, num2) =>
num1 + num2;
se tiver mais de uma linha usa
chaves e return(if e else pede
return)
ex: const
somaNumerosPequenos = (num1,num2)
=> {
if (num 1 \\ num > 10) {
                    return "somente
números de 1 a 9"     }else{
                              return num
+ num2;                       }
}
```

- hoisting: arrow function se comporta como expressão
- omo retornar informações da função, utilizando o return, lembrando que o console.log() apenas mostra a informação no terminal e não para outras partes do código.
- A utilidade dos argumentos, já que com eles podemos passar variáveis para as funções poderem usar os valores.
- Que com o *hoisting* o JavaScript analisa todo o código procurando por variáveis declaradas com var e funções para trazer tais declarações para o início do código.
- Expressões de função, uma maneira diferente de montar funções usando variáveis do tipo const e chamando-as pelo nome. Lembrando que é necessário que o programa passe pela variável antes de podermos chamá-la, já que não há suporte à *hoisting*.

- *Arrow function*, uma função declarada de maneira mais compacta usando uma const. A *arrow function* também não tem suporte à *hoisting*.
-