



Aprenda a programar

Dia 2 – Aprendendo a programar e
adquirindo o hábito com GitHub, parte 2

O que veremos

Estruturas condicionais e operadores relacionais:

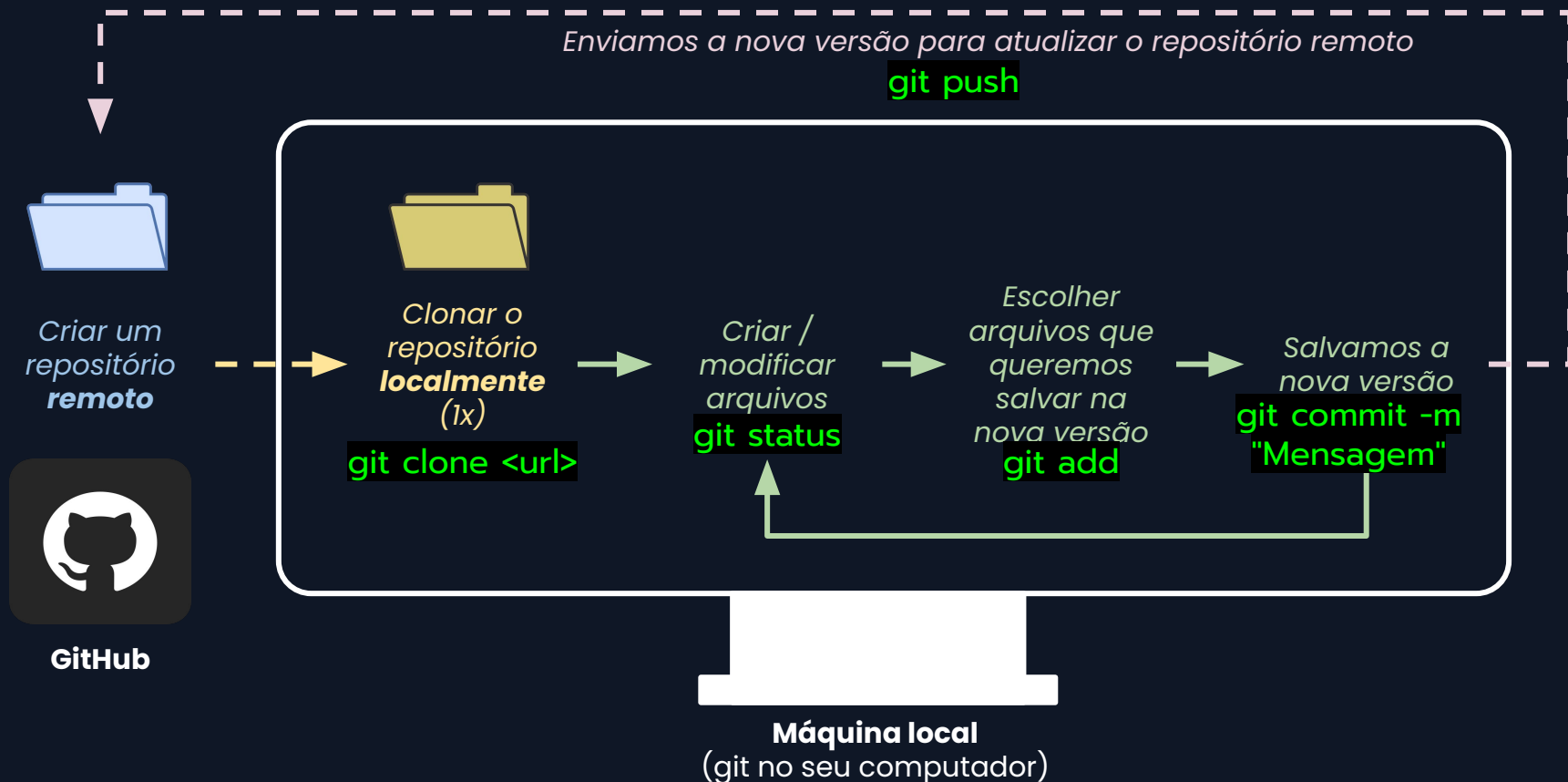
a 'inteligência' por trás de tudo

Estruturas de repetição:

a magia de processar e iterar

Relembrando a aula passada...

Introdução ao git e ao GitHub



Introdução ao git e ao GitHub



Comandos de programação

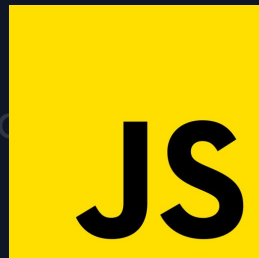
Transformando um código em um programa/aplicação



Comandos de programação

Transformando um código em programa/aplicação

Onde entra o JavaScript nisso?



Instruções/comandos
Lógica

Código fonte



JavaScript é a **linguagem** que você está se baseando para escrever o seu código.

Transformando o código fonte em

O código que você escrever deve

obedecer as regras da linguagem JavaScript.



Programa executado



Commando no terminal:

```
node meucodigo.js
```

Comandos de programação

O que constitui um programa de computador?

Um programa constitui de uma sequência de comandos que o computador sempre executa em ordem.



Comando 1

Comando 2

Comando 3

Comando 4

Comando 5

Comando 6

Comandos de programação

O que constitui um programa de computador?

Um programa constitui de uma sequência de comandos que o computador sempre executa em ordem.



Comando 1	1	<code>let nome = "Maria"</code>
Comando 2	2	<code>let email = "maria@maria.com"</code>
Comando 3	3	<code>let profissao = "publicitária"</code>
Comando 4	4	<code>let idade = 33</code>
Comando 5	5	<code>let viagemDosSonhos = "Bali"</code>
Comando 6	6	
Comando 7	7	<code>let mensagem = "Olá, " + nome + "! Somos aqui da agência de turismo Viagem dos Sonhos. Estamos te escrevendo este email, pois acabamos de confirmar as compras das passagens aéreas para " + viagemDosSonhos + ". Confirma pra gente alguns dados? A sua idade é " + idade + ", você é " + profissao + " e o seu email é " + email + "? Ficamos no aguardo. Muito obrigado e boa viagem! :D"</code>
Comando 8	8	
Comando 9	9	<code>console.log(mensagem)</code>

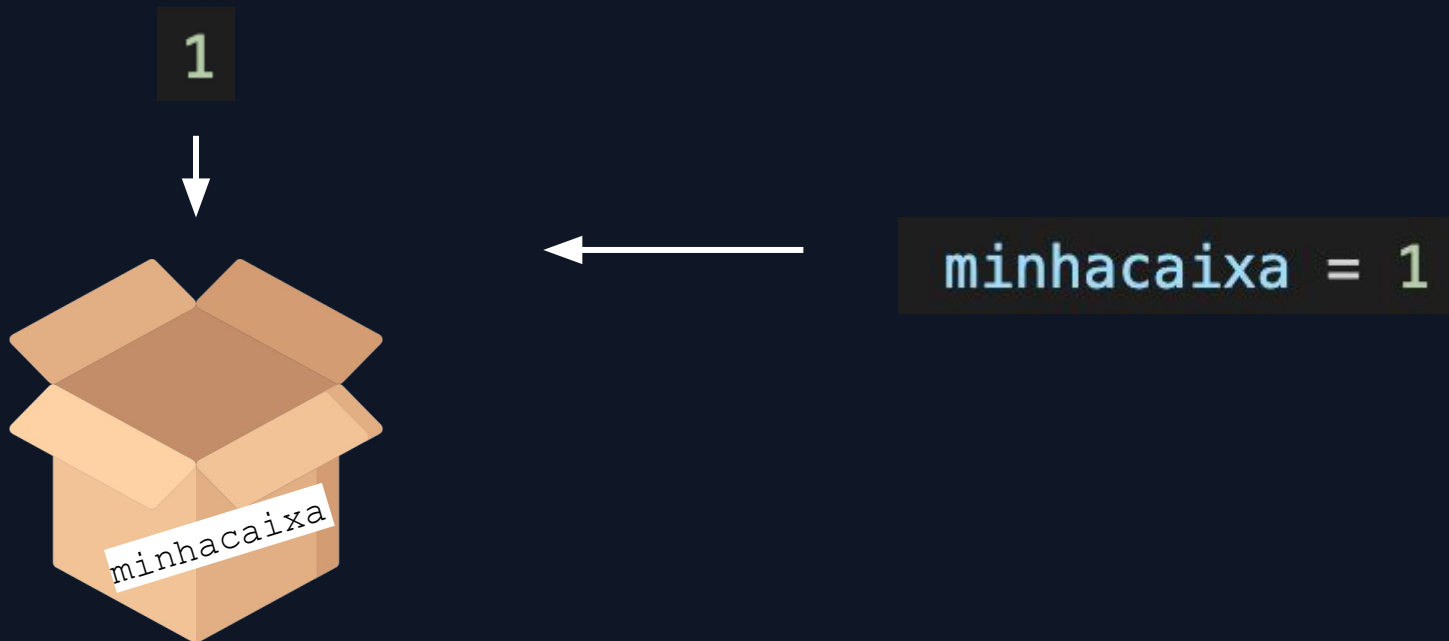
Comando de declaração de variável



```
let minhacaixa
```

Tipos de dados, variáveis e memória

Comando de atribuição



Tipos de dados, variáveis e memória

Comando de atribuição



Tipos de dados, variáveis e memória

As variáveis podem receber dados dos **tipos**:

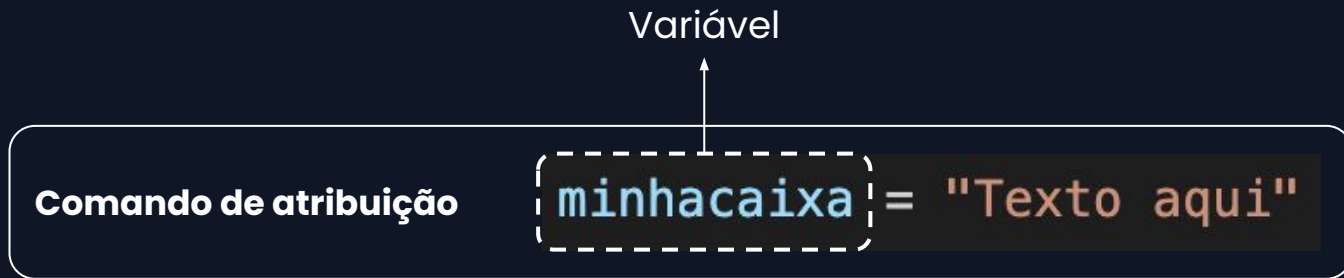
- **Strings (texto)** - "Instrutor Tiago"
- **Number (número)** - 12
- **Booleans** - podem ser **true** ou **false**
- **Arrays (Listas)** - Listas de qualquer tipo de variável (inclusive listas de listas)
- **Mapas** - pares de chave e valor

No curso só trabalharemos com estes tipos de dados



Tipos de dados, variáveis e memória

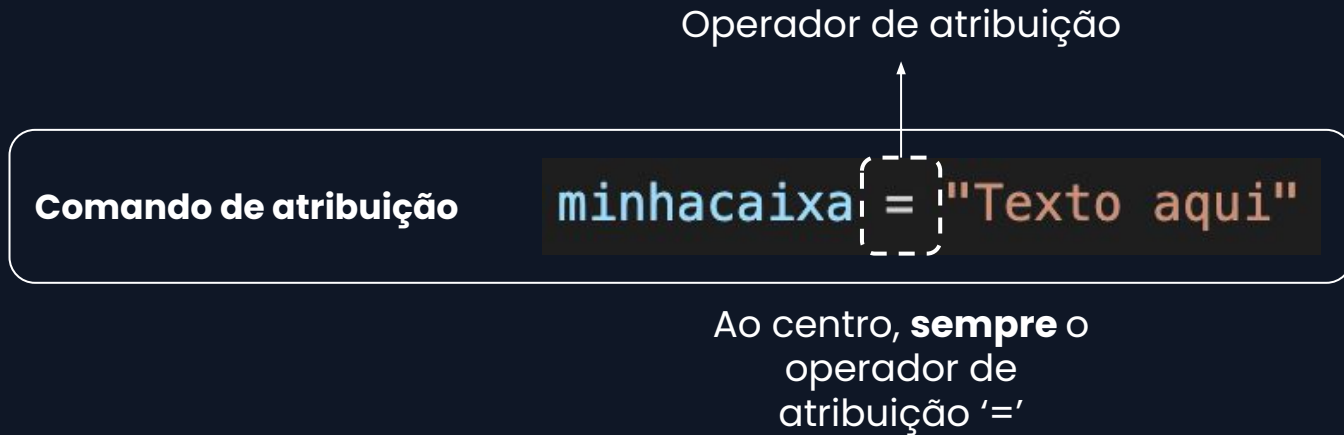
Entendendo melhor Comandos de atribuição



Do lado esquerdo, **sempre**
uma variável

Tipos de dados, variáveis e memória

Entendendo melhor Comandos de atribuição



Tipos de dados, variáveis e memória

Entendendo melhor Comandos de atribuição

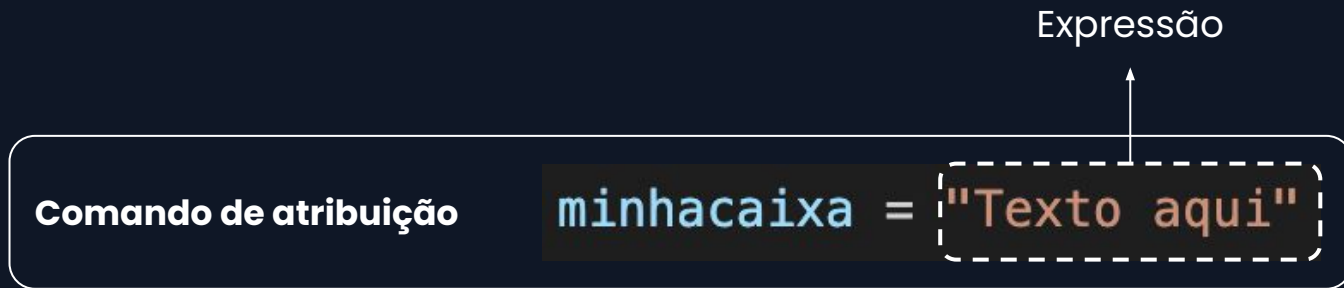
Comando de atribuição

```
minhacaixa = "Texto aqui"
```

Expressão

No lado direito,
sempre uma
expressão

Expressão **sempre** resulta em um valor final.



Aqui o valor final
dessa expressão é
"Texto aqui"

Ou seja, uma string.

Expressão **sempre** resulta em um valor final.

```
1 | let x = 1  
2 | let y = x
```

Expressões

Expressão **sempre** resulta em um valor final.

```
1 | let x = 1  
2 | let y = x
```



Expressões

Expressão **sempre** resulta em um valor final.

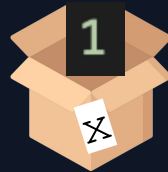
```
1 | let x = 1
2 | let y = x
```



Expressões

Expressão **sempre** resulta em um valor final.

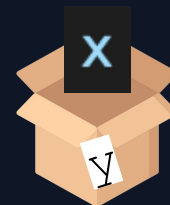
```
1 | let x = 1  
2 | let y = x
```



Expressões

Expressão **sempre** resulta em um valor final.

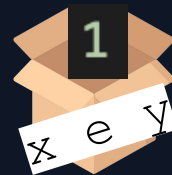
```
1 | let x = 1
2 | let y = x
```



Expressões

Expressão **sempre** resulta em um valor final.

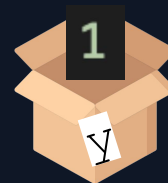
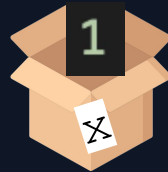
```
1 | let x = 1
2 | let y = x
```



Expressões

Expressão **sempre** resulta em um valor final.

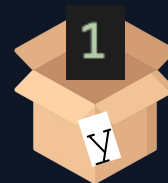
```
1 | let x = 1  
2 | let y = x
```



Expressões

Expressão **sempre** resulta em um valor final.

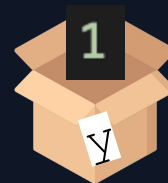
```
1 | let x = 1  
2 | let y = 1
```



Expressões

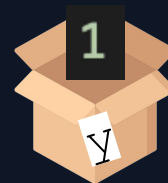
Expressão **sempre** resulta em um valor final.

```
1 | let x = 1  
2 | let y = x
```



Expressão **sempre** resulta em um valor final.

Exercite **sempre** o poder de abstração da sua mente e imagine o que está acontecendo com os valores que vão sendo armazenados nas variáveis



Expressões aritméticas (+, -, *, /)

É aceita qualquer uma das operações básicas da matemática



+

soma

-

subtração

*

multiplicação

/

divisão

Expressões aritméticas (+, -, *, /)

Comando de atribuição

```
minhacaixa = 1+2*3+4
```

Expressão aritmética

Aqui o valor final dessa expressão é 11

Ou seja, o valor armazenado em **minhacaixa** será 11.

Expressões aritméticas (+, -, *, /)

Comando de atribuição

`minhacaixa = (1+2)*(3+4)`

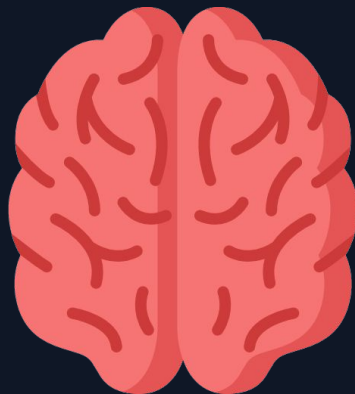
Expressão aritmética

Aqui o valor final dessa expressão é 21

Ou seja, o valor armazenado em **minhacaixa** será 21.

**Estruturas condicionais e
operadores relacionais:**
a 'inteligência' por trás de tudo

Condições são a “inteligência” do computador!



Não é inteligência artificial... isso é papo para outro dia.



Comando A



Comando B



Comando C

Sabemos que o computador é muito bom em executar sequência de comandos

Mas seria bom se ele pudesse ser capaz de tomar algumas decisões sozinho, né?

>_ Comando A

>_ Comando B

>_ Comando C

Imagine que depois do comando C, nós quiséssemos ter duas opções de comandos possíveis e o computador possa **decidir** sobre qual caminho seguir?

```
graph TD; A[Comando A] --> B[Comando B]; B --> C[Comando C]; C --> D[Comando D];
```

>_ Comando A

>_ Comando B

>_ Comando C

>_ Comando D

Podemos definir um caminho em que o computador **decide** executar um comando D

>_ Comando A

>_ Comando B

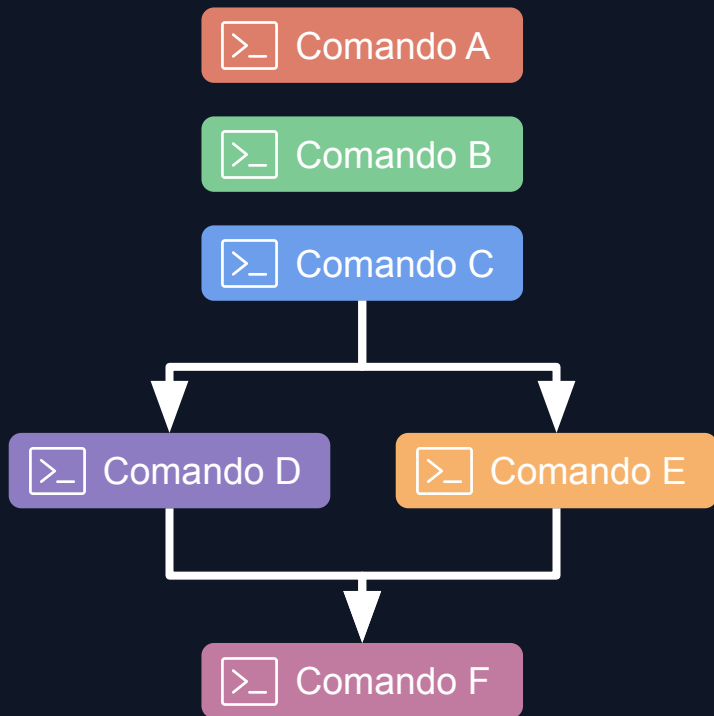
>_ Comando C



>_ Comando D

>_ Comando E

E podemos definir um outro caminho em que o computador **decide** executar um comando E



Depois disso, o computador volta a executar o código na sequência

Como criar essa “inteligência”?

Usando estruturas condicionais

Se (condição)
é verdade?

Faz uma coisa

sim

Faz outra coisa

não

>_ Comando A

>_ Comando B

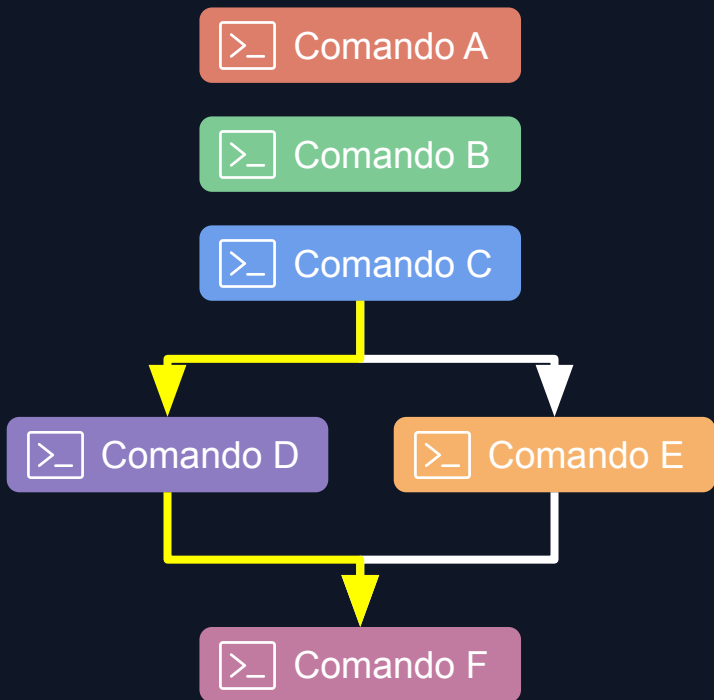
>_ Comando C

>_ Comando D

>_ Comando E

>_ Comando F

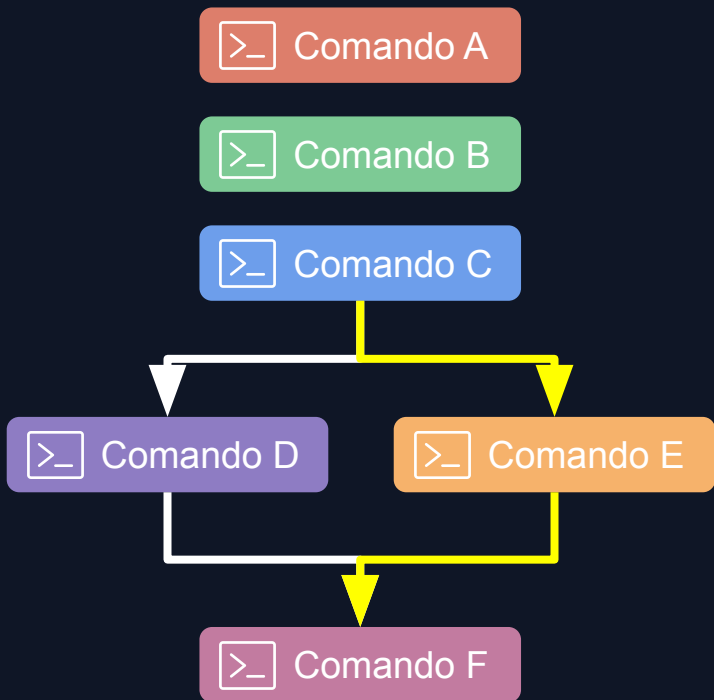
Se (**condição**) verdade ?



Se (**condição**) verdade ?

Sim

[Comando D]



Se (**condição**) verdade ?

Sim

[Comando D]

Não

[Comando E]

Se nosso programa
fosse um
jogo de poker
online

Se (**condição**) verdade ?

Sim



Comando D

Não



Comando E

A **condição** pra
jogar é ter pelo
menos 18 anos



Se nosso programa
fosse um
jogo de poker
online

Se (**idade** \geq **18**) verdade ?

Sim `console.log("Pode jogar")`

Não `console.log("Acesso negado!")`

A **condição** pra
jogar é ter pelo
menos 18 anos



```
if (idade >= 18) {  
    console.log("Pode jogar");  
} else {  
    console.log("Acesso negado!");  
}
```

Se (**idade >= 18**) verdade ?

Sim `console.log("Pode jogar")`

Não `console.log("Acesso negado!")`


```
if (idade >= 18) {  
    console.log("Pode jogar");  
} else {  
    console.log("Acesso negado!");  
}
```

Se (**idade >= 18**) verdade ?

Sim `console.log("Pode jogar")`

Não `console.log("Acesso negado!")`

Mas, o que isso faz e como ele entra dentro do if?

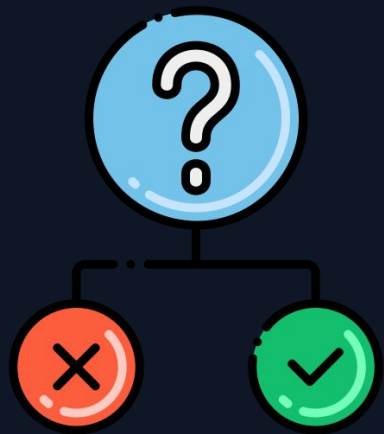


```
if (idade >= 18) {  
    console.log("Pode jogar");  
} else {  
    console.log("Acesso negado!");  
}
```

Expressões relacionais ($==$, $!=$, $>$, $<$, $>=$, $<=$)

Expressões relacionais ($=$, \neq , $>$, $<$, \geq , \leq)

Expressões relacionais compara valores e retorna **true** ou **false**

 $>$

maior que

 $<$

menor que

 \geq

maior ou igual a

 \leq

menor ou igual a

 \neq

diferente

 $=$

igual a

Expressões relacionais (==, !=, >, <, >=, <=)

Expressão relacional

Lembra que expressão
sempre resulta em um valor final?

```
if (idade >= 18) {  
    console.log("Pode jogar");  
} else {  
    console.log("Acesso negado!");  
}
```

Expressões relacionais (==, !=, >, <, >=, <=)

Lembra que expressão
sempre resulta em um valor final?

```
if ( true ) {  
    console.log("Pode jogar");  
} else {  
    console.log("Acesso negado!");  
}
```

Expressões relacionais (==, !=, >, <, >=, <=)

Lembra que expressão
sempre resulta em um valor final?

```
if ( false ) {  
    console.log("Pode jogar");  
} else {  
    console.log("Acesso negado!");  
}
```

Expressões relacionais (==, !=, >, <, >=, <=)

Expressão relacional

Comando de atribuição

```
minhacaixa = 1 < 3
```

Expressões relacionais (==, !=, >, <, >=, <=)

Comando de atribuição

```
minhacaixa = 1 < 3
```

Expressão relacional

Valor final dessa expressão é **true**

Expressões relacionais (==, !=, >, <, >=, <=)

Comando de atribuição

```
minhacaixa = 1 < 3
```

Expressão relacional

Valor armazenado
por **minhacaixa**
será **true**

Expressões relacionais (==, !=, >, <, >=, <=)

Expressão relacional

Comando de atribuição

`minhacaixa = 1 == 1`

Expressões relacionais (==, !=, >, <, >=, <=)

Comando de atribuição

```
minhacaixa = 1 == 1
```

Expressão relacional

Valor final dessa
expressão é **true**

Expressões relacionais (==, !=, >, <, >=, <=)

Comando de atribuição

```
minhacaixa = 1 == 1
```

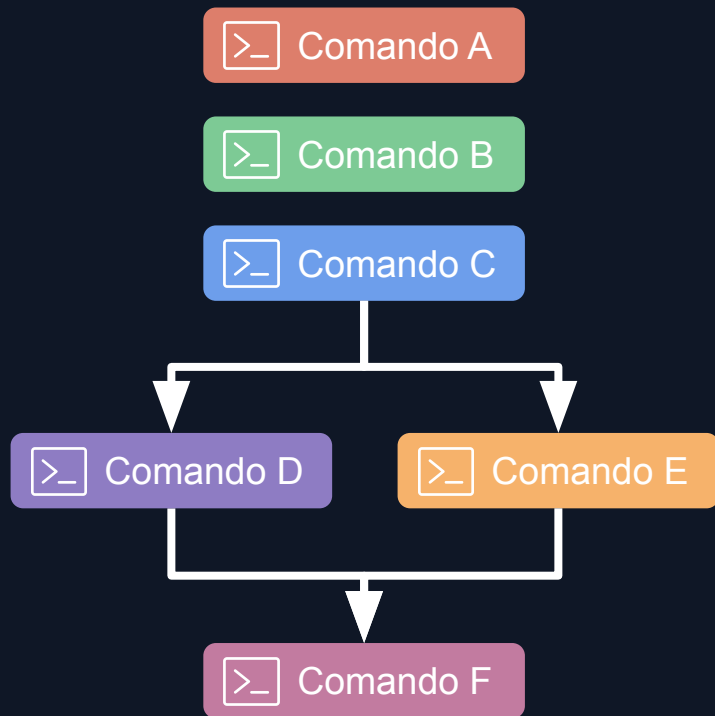
Expressão relacional

Valor armazenado
por **minhacaixa**
será **true**

Bora praticar um pouco as estruturas condicionais!

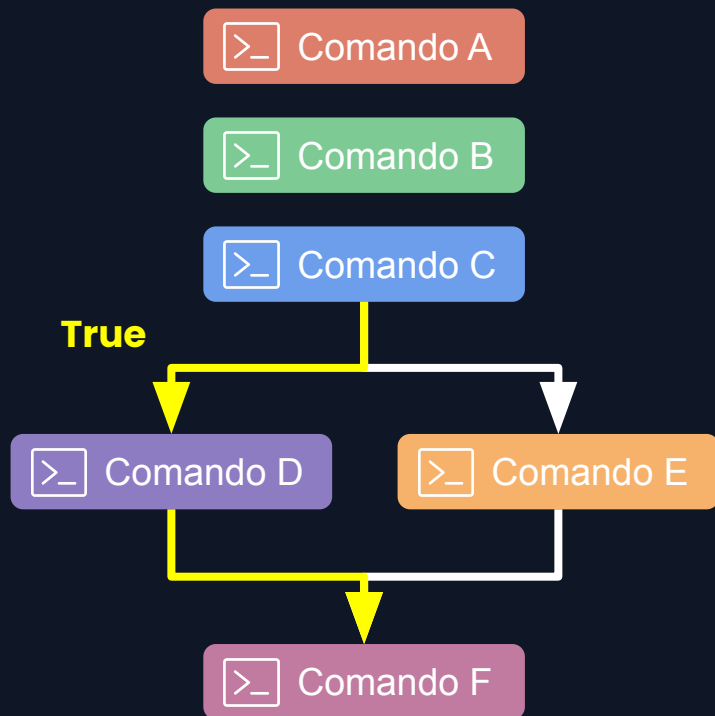
Estruturas de repetição:
a magia de processar e iterar

O que é iterar?



Em estruturas **condicionais**, aprendemos a mudar o fluxo de um código.

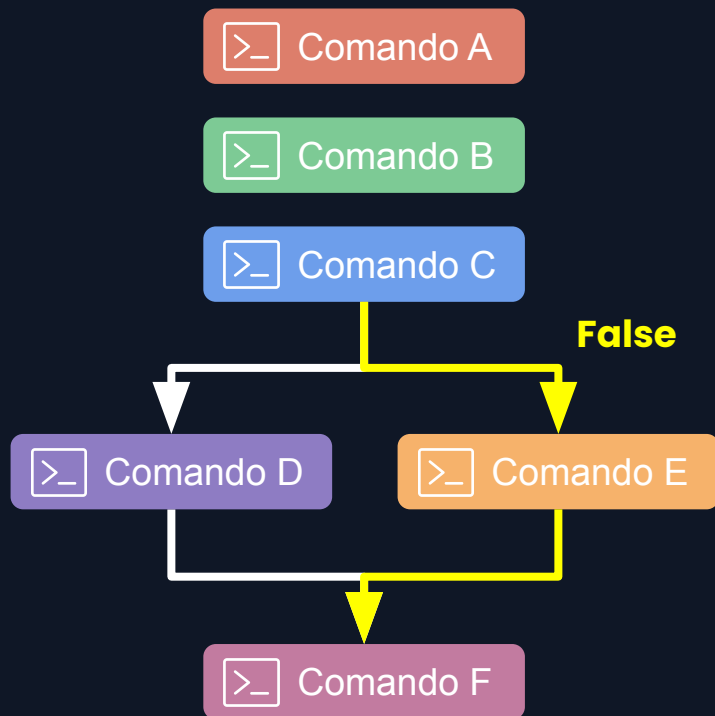
O que é iterar?



Em estruturas **condicionais**, aprendemos a mudar o fluxo de um código.

Conseguimos definir um caminho diferente para o computador executar

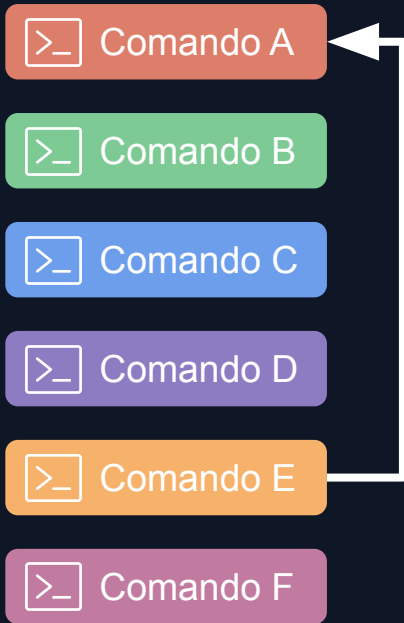
O que é iterar?



Em estruturas **condicionais**, aprendemos a mudar o fluxo de um código.

Conseguimos definir um caminho diferente para o computador executar

O que é iterar?



?

E se pudéssemos fazer com que, ao invés de executar um novo comando, o computador voltar a executar os comandos anteriores?

Ou seja, **repetir** os comandos anteriores. É possível?

O que é iterar?

Iterar é **repetir** um conjunto de comandos de um código.

Mas porque precisamos repetir?



Jogo de luta: enquanto o adversário estiver “vivo”, vou bater nele.



Fazer um ovo mexido: enquanto o ovo não estiver misturado e pronto, vou mexer ele.



Um cavalo comendo capim: enquanto não estiver saciado, vou comer capim.

Como criar esse “loop”, essa repetição?

Usando estruturas de repetição

Enquanto (**condição**) *é verdade?*

**Repita este conjunto
de comandos**

sim

Pule estes comandos

não

>_ Comando A

>_ Comando B

>_ Comando C

>_ Comando D

>_ Comando E

>_ Comando F

>_ Comando A

Sim

Não

>_ Comando A

>_ Comando B

>_ Comando C

>_ Comando D

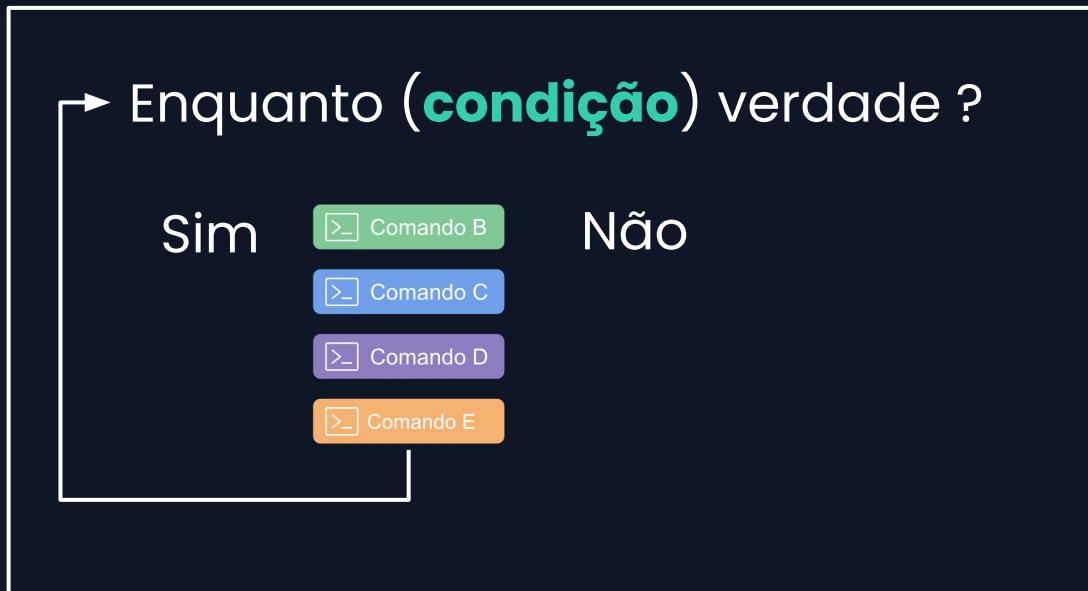
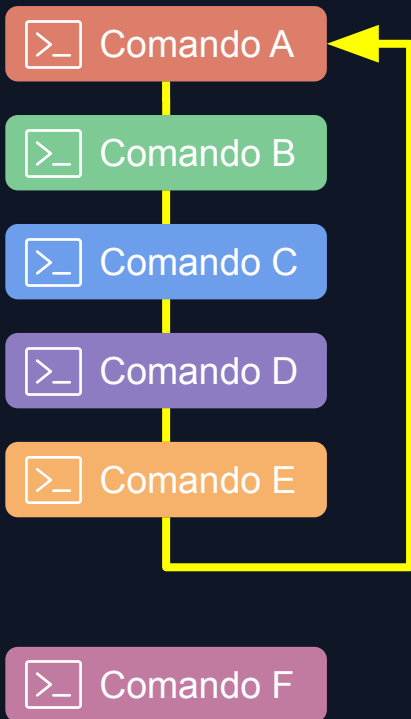
>_ Comando E

>_ Comando F

Enquanto (**condição**) verdade ?

Sim

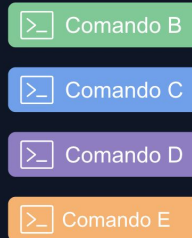
Não





Enquanto (**condição**) verdade ?

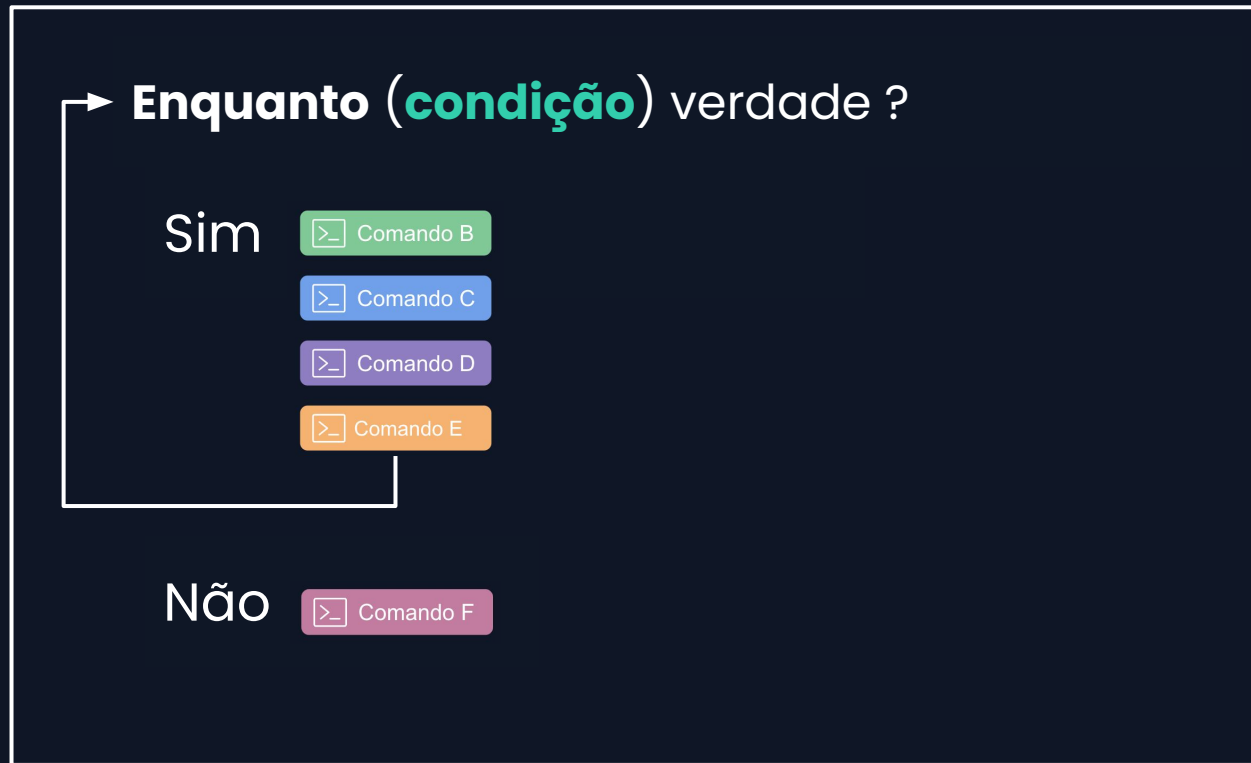
Sim



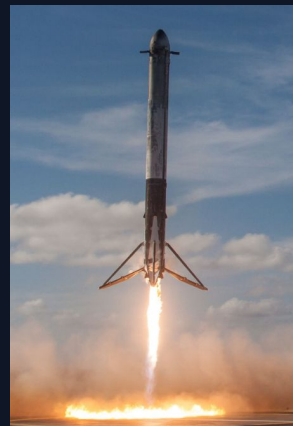
Não



Se nosso programa fosse um foguete da SpaceX...



A **condição** é foguete **não** estar pousado



Se nosso programa fosse um foguete da SpaceX...



A **condição** é foguete **não** estar pousado



Se nosso programa fosse um foguete da SpaceX...

→ Enquanto (**distanciaDoChao > 0**) verdade ?

Sim

```
garantirEstabilidade()  
distanciaDoChao -= 1  
garantirEquilibrio()  
controlarPotenciaDoMotor()
```

Não

> Comando F

Se isso for **verdade**, ou seja, o foguete **não** estiver pousado, ele continua efetuando as manobras



Se nosso programa fosse um foguete da SpaceX...

Enquanto (**distanciaDoChao > 0**) verdade ?

Sim

```
garantirEstabilidade()
distanciaDoChao -= 1
garantirEquilibrio()
controlarPotenciaDoMotor()
```

Não

```
console.log("Foguete pousou!")
```

Se isso **não** for verdade, ou seja, o foguete estiver pousado, mostramos no console "Foguete pousou"



Se nosso programa fosse um foguete da SpaceX...

Enquanto (**distanciaDoChao > 0**) verdade ?

Sim

```
garantirEstabilidade()  
distanciaDoChao -= 1  
garantirEquilibrio()  
controlarPotenciaDoMotor()
```

Não

```
console.log("Foguete pousou!")
```

Agora vamos
traduzir para
JavaScript

```
while (distanciaDoChao > 0) {  
    garantirEstabilidade()  
    distanciaDoChao--  
    garantirEquilibrio()  
    controlarPotenciaDoMotor()  
}  
  
console.log("Foguete pousou!")
```

Enquanto (distanciaDoChao > 0) verdade ?

Sim

```
garantirEstabilidade()  
distanciaDoChao -= 1  
garantirEquilibrio()  
controlarPotenciaDoMotor()
```

Não

```
console.log("Foguete pousou!")
```

```
while (distanciaDoChao > 0) {  
    garantirEstabilidade()  
    distanciaDoChao--  
    garantirEquilibrio()  
    controlarPotenciaDoMotor()  
}
```

```
console.log("Foguete pousou!")
```

```
while (condicao) {
    // comando1
    // comando2
    // comando3
    // ...
}
// condição falsa, sai do loop
```

Qualquer problema lógico **solucionável** no mundo
pode ser resolvido com

Declaração de variáveis

Comando de atribuição

Comandos de repetição

Comandos condicionais

'Toda' linguagem de programação possui:

Declaração de variáveis

```
let variavel
```

Comando de atribuição

```
variavel = 1
```

Operadores aritméticos

```
+ SOMA
```

```
- SUBTRAÇÃO
```

```
* MULTIPLICAÇÃO
```

```
/ DIVISÃO
```

Operadores relacionais

```
> MAIOR QUE
```

```
< MENOR QUE
```

```
== IGUALDADE
```

```
>= MAIOR OU IGUAL
```

```
<= MENOR OU IGUAL
```

'Toda' linguagem de programação possui:

Comandos condicionais

```
if (/* expressão */) {
    // caso verdadeiro
}
```

```
if (/* expressão */) {
    // caso verdadeiro
} else {
    // caso falso
}
```

Palavras/símbolos reservados

```
if
else
let
+
-
/
*
(
)
{
}
```

...

'Toda' linguagem de programação possui:

Comandos de repetição

```
while (/* condição */) {
    // caso verdadeiro
}
```

Palavras/símbolos reservadas

```
if
else
let
+
-
/
*
(
)
{
}
```

...

Mas cada linguagem possui **sua sintaxe**

JS

```
1 console.log("Hello World!");
```

Mas cada linguagem possui **sua sintaxe**



```
1 print("Hello World")
```

Mas cada linguagem possui **sua sintaxe**



```
1 <?php echo "Hello, World";
```

Mas cada linguagem possui **sua sintaxe**



```
1 using System;
2
3 class Program
4 {
5     static void Main(string[] args)
6     {
7         Console.WriteLine("Hello, world!");
8     }
9 }
```

Mas cada linguagem possui **sua sintaxe**



```
1 class HelloWorldApp {  
2     public static void main(String[] args) {  
3         System.out.println("Hello World!");  
4     }  
5 }
```


Mas cada linguagem possui **sua sintaxe**



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world\n");
6 }
```

Tarefa do Dia 3 vamos disponibilizar **amanhã!**

Prazo final de envio da tarefa do Dia 3: domingo (29/08)

Gravação e slides da aula **ainda hoje.**