



Bem-vindos
à disciplina

ALGORITMOS E PROGRAMAÇÃO I



Você vai ver...

Fundamentos da Programação

TIPOS DE DADOS

Tipos
Variáveis
Classes
Arrays
Matrizes

FUNÇÕES

Operadores
Funções
Composição
Abstração
Funções Recursivas
Funções de Alta Ordem

CONTROLE DE FLUXO

Comandos Condicionais
Comandos de Repetição
Encadeamento de Comandos

...e todos conceitos que envolvem a criação de um software robusto para alta escala



Olá, sou Bruno de Oliveira

brunodeoliveira.22.10@gmail.com

Graduado em
Sistemas de Informação

Pós-Graduado em
BigData

Mestre em
Tecnologias da Inteligência e Design Digital

No mercado de software há
14 anos

Certificado em
7 certificações Microsoft

Vencedor de hackatons

- SPTrans
- Digital Innovation One
- VAI TeC

Empreendedor

- Cadê o Ônibus
- DevMonk
- Consultoria de Software

...e mais alguns na gaveta



ENCONTRO 01

Introdução a Linguagem, Lógica e Pensamento



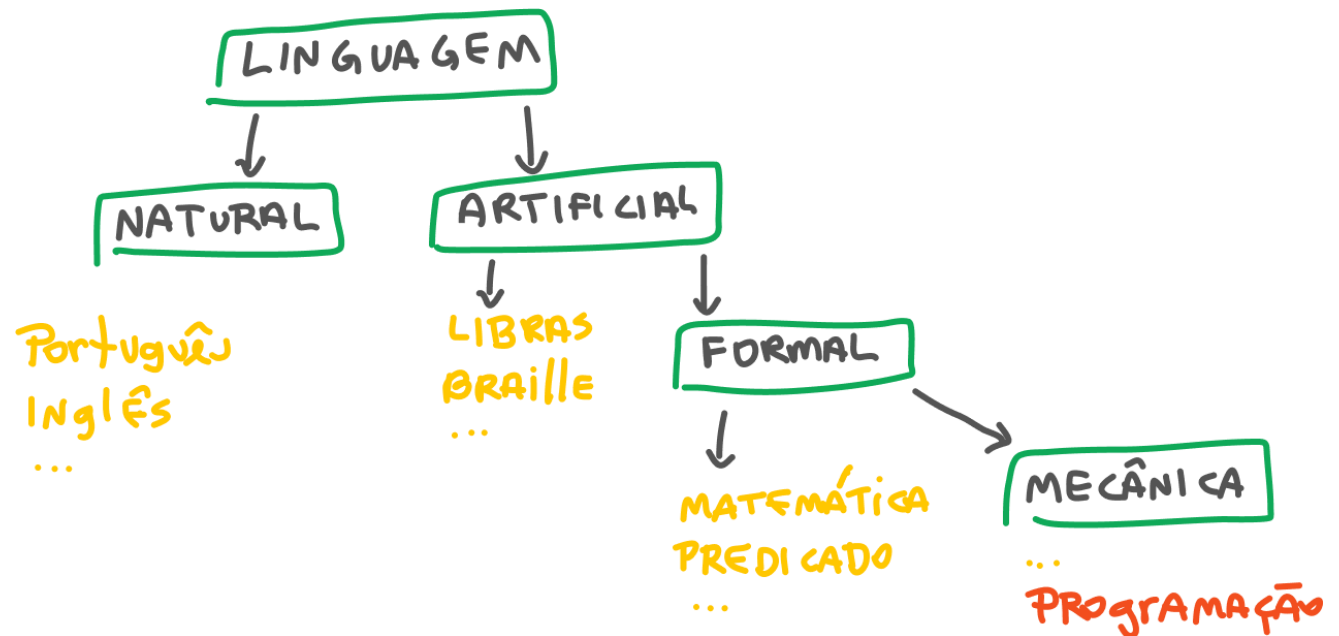
Agenda

- Categorias da Linguagem
- Elementos da Comunicação
- O que é Programar?
- Caracterizando a Linguagem



Categorias de Linguagem

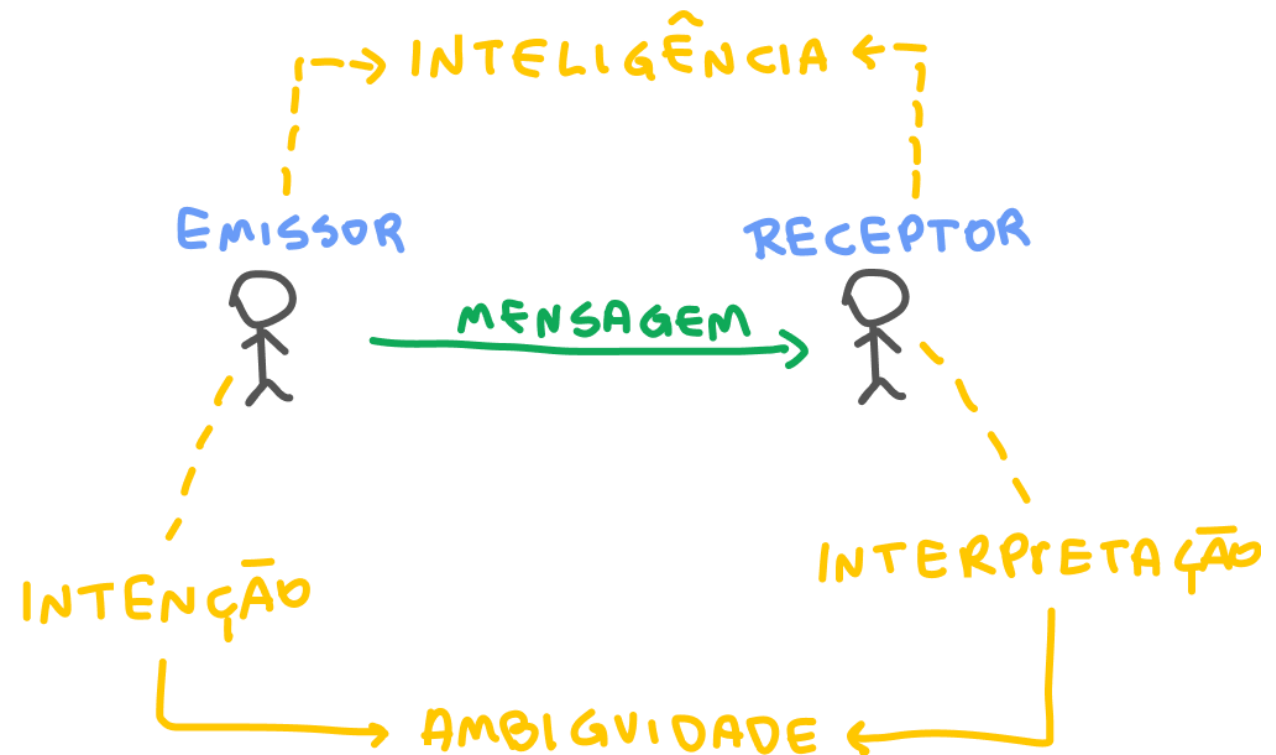
A linguagem de programação encontra-se localizada nas **linguagens mecânicas, formais, artificiais**. Esse contexto já aponta o nível de rigor e precisão exigidos para os raciocínios feitos nesse ambiente.





Elementos da Comunicação

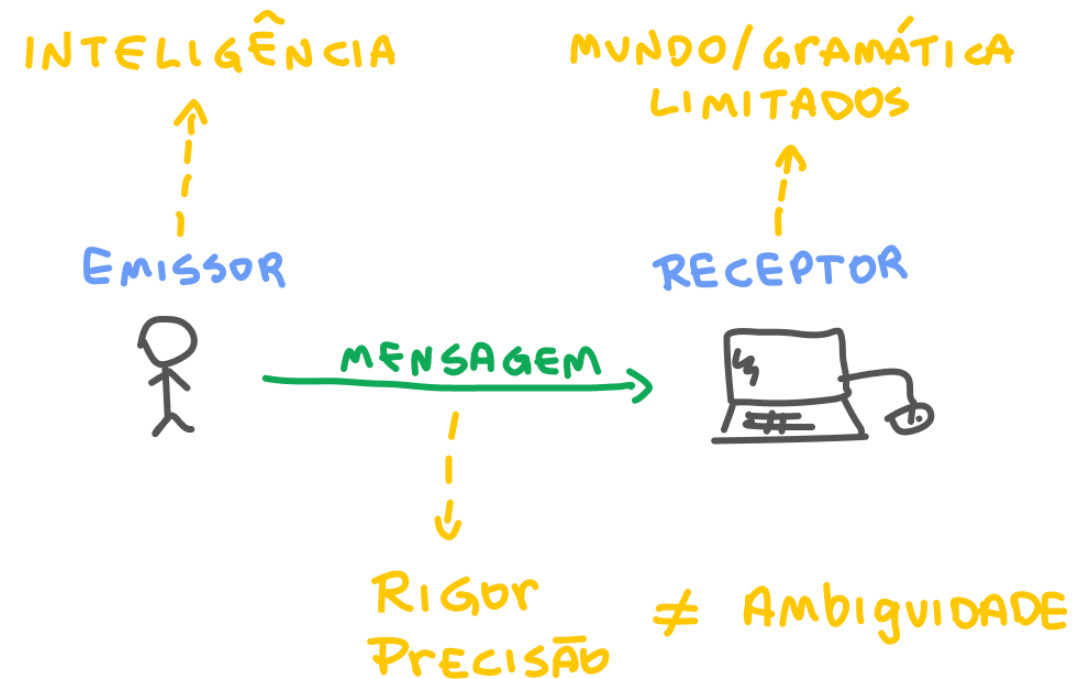
No Mundo Natural, o homem se comunica com outro homem, dois seres **inteligentes** que trocam mensagens para que se compreendam. Nesse ambiente, a linguagem natural pode apresentar ruídos na comunicação devido a **intencionalidade** do emissor não corresponder a **interpretabilidade** do receptor.





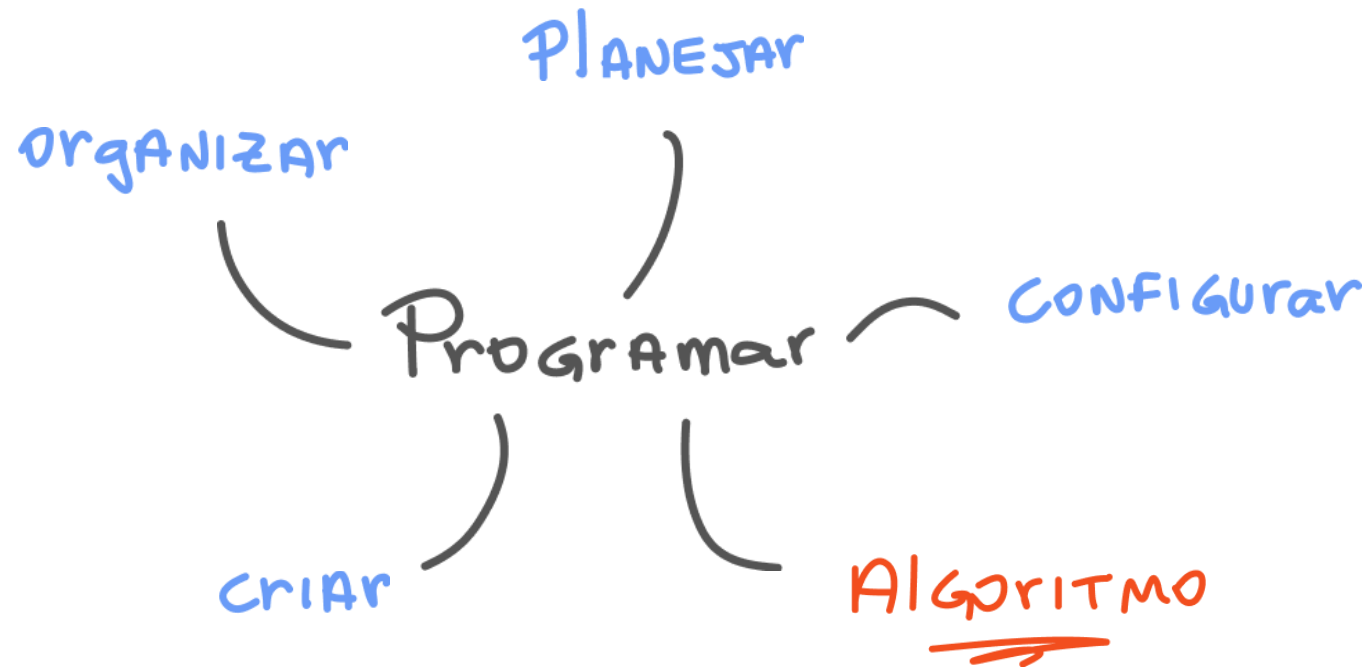
Elementos da Comunicação

No Mundo Digital, o homem se comunica com a máquina, que é programada para compreender uma **gramática limitada**, não sensível ao contexto. Assim o homem deve imperar sobre a máquina no sentido de comandá-la, **dar-lhe ordens**, as quais precisam ser **precisas em bem formadas** para que possa ser compreendida por sua limitação e falta de inteligência.





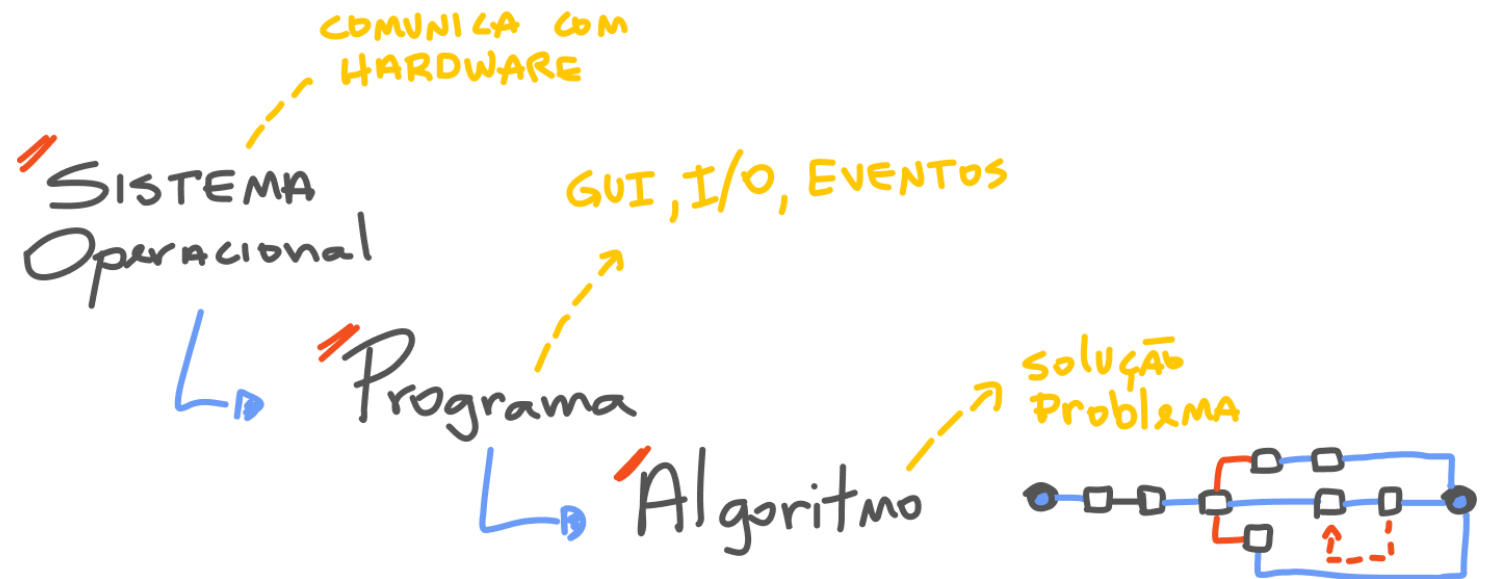
O que você entende por Programar?





O Ato de programar

O ato de programar consiste na habilidade de criar algoritmos, ou seja, uma sequência de passos **finita, precisa e não ambígua**, com início, meio e fim, que pode ser executada mecanicamente, tendo por finalidade, a **solução de um problema**.

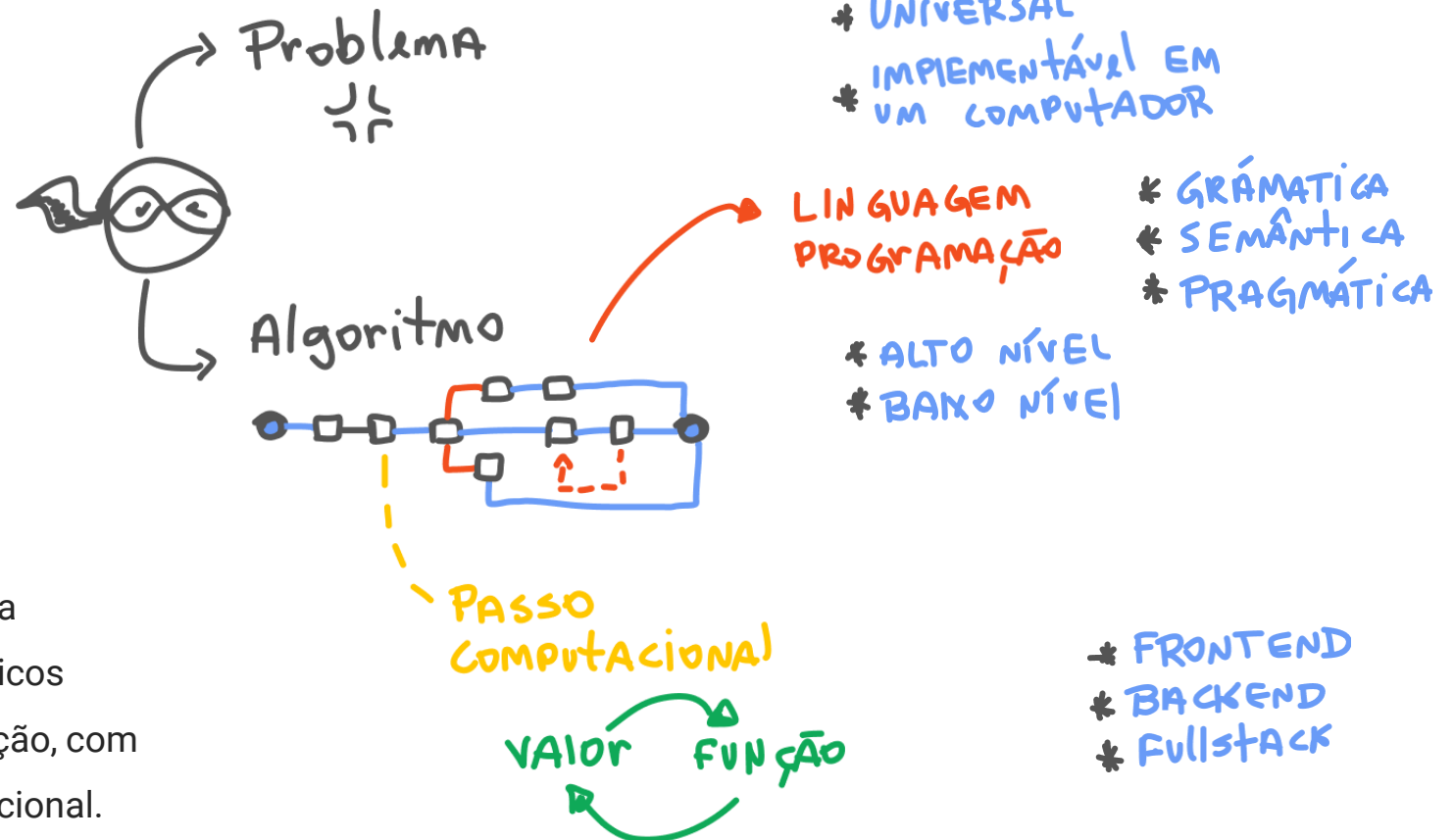




Tudo começa por um problema

A partir de um **problema** que deve ter sua solução por meio de um computador, o programador representa sua abstração através de um algoritmo, o qual é **implementado** através de uma linguagem, que por sua vez possui suas características próprias, mas que deve seguir alguns requisitos para ser considerada “de programação”.

Podemos considerar Lógica de Programação, a capacidade de combinar os elementos linguísticos estabelecidos nos Fundamentos da Programação, com a finalidade de resolver um problema computacional.





ENCONTRO 02

Elementos Primitivos e Compostos



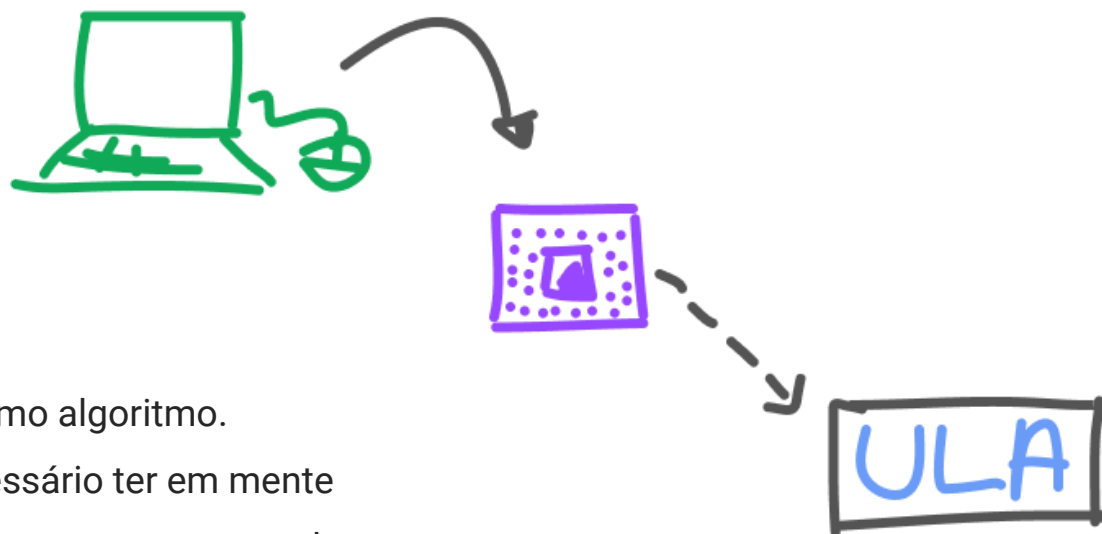
Agenda

- O que um computador Faz?
- Elementos Primários e Compostos
- Estrutura TFR
- Jornada de um Programador



O que um computador Faz?

Um computador *computa* informações, isto é, realiza cálculos. A ULA é o componente do processador responsável pela realização dos cálculos aritméticos e lógicos. Através desse ponto de vista, sequenciar os cálculos necessários para a solução de um problema, é o que já definimos como algoritmo. Resulta que para criar programas de computador é necessário ter em mente três elementos essenciais: (1) Valores, (2) Operações que computam os valores, (3) Sequência das operações.





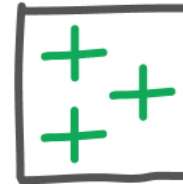
Elementos Primários e Compostos

Na linguagem de programação e em todas outras áreas do conhecimento são determinados **elementos primários**, dos quais geram-se **elementos mais complexos**. A habilidade de perceber os elementos primários, combiná-los e criar novos elementos é a base do **pensamento formal, sistêmico e rigoroso**.

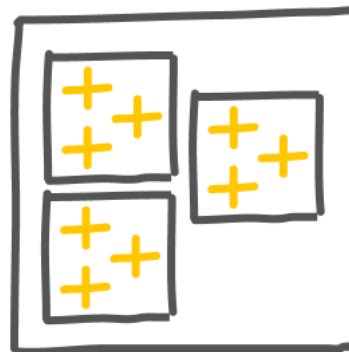
Soma



Multiplicação



Potenciação



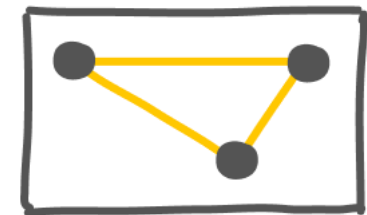
Ponto



Reta



Plano





TFR: Graus de Complexidade

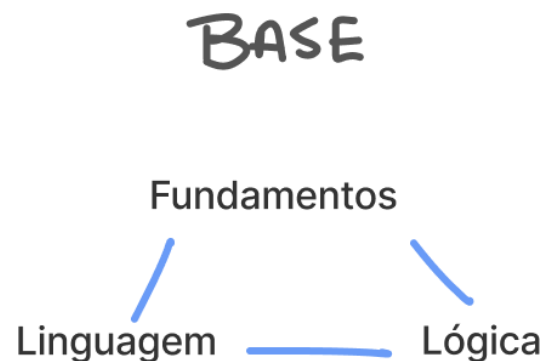
A estrutura TFR apresenta os graus de complexidade dos elementos envolvidos no “ato de programar”, baseando-se nos princípios de composição e abstração.

Type	Function	Route
1 Primitives	1 Primitives	1 Simple
2 Simple Structures	2 High level Functions	2 Conditional
3 Compound Structures	3 Simple Procedures	3 Compound Conditional
4 Recursives	4 Compound procedures	4 Looping
	5 Packages	5 Compound Looping
	6 Recursive	
	7 High order	

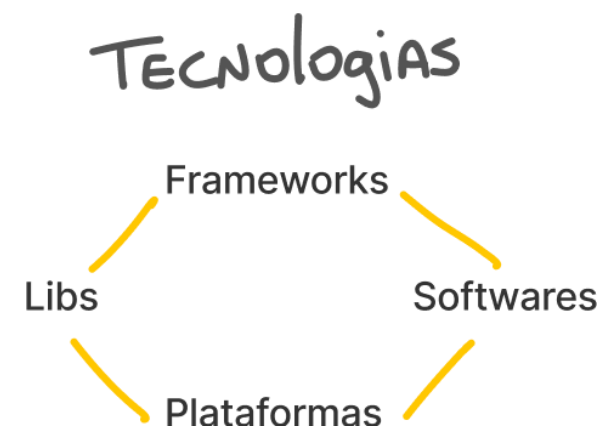
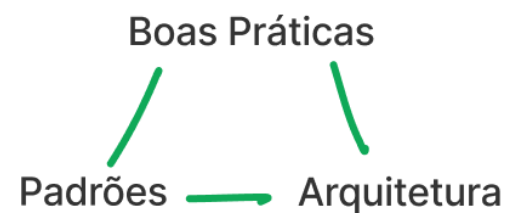


Do estágio à Senioridade

Em algum ponto de nossa carreira nos perguntamos: **“Do que preciso para me tornar Senior?”**. Essa pergunta abre espaço para discutirmos a jornada do programador contemporâneo, cheio de informações, canais, blogs e todo acesso que a internet oferece. Quem são esses atrás da telinha? **Como comparar um livro que contém as ideias de um escritor pesquisador estudioso com um vídeo de dez minutos com cortes e bem editado?**



AMADURECIMENTO





ENCONTRO 03

As ideias bases: Valores e Operadores



Agenda

- Representações
- Valores
 - Primitivos
 - Compostos
- Funções
 - Primitivas
 - Compostas



Representações

É imperioso representar **dados de diferentes espécies** enquanto codificamos um programa. Da mesma forma, é necessário representar **operações para manipular esses dados**.

Nas linguagens de programação de alto nível, dados e operações podem ser divididos em dois grupos: **primitivos e compostos**, sendo o segundo uma categoria com subdivisões que veremos ao decorrer do curso. Em geral, os dados primitivos são representados por **números, caracteres e a ideia de verdadeiro e falso**. As operações primitivas são responsáveis por manipular os dados primitivos geralmente expressadas por **operações aritméticas, relacionais e lógicas**.

Handwritten examples of data and operations:

Data (Primitives):

- String: Bruno
- Integer: 32
- Boolean: NÃO (No), Sim (Yes)
- Float: 1.85
- Character: A
- Date: 22/10/89

Operations (Primitives):

- Relational: >
- Arithmetic: ÷, ×, +, -
- Logical: &&



Parte 01

Valores



Roteiro de Treino

nas linguagens estáticas – Java

Objetivos

- Conhecer os valores primitivos;
- Conhecer alguns valores compostos;
- Perceber o rigor da linguagem;
- Conhecer a formalidade das representações;
- Introduzir regras para representar números;
- Introduzir regras para representar textos;
- Introduzir regras para representar datas;
- Introduzir regras para representar booleanos;
- **Apontar os tipos primitivos e compostos**

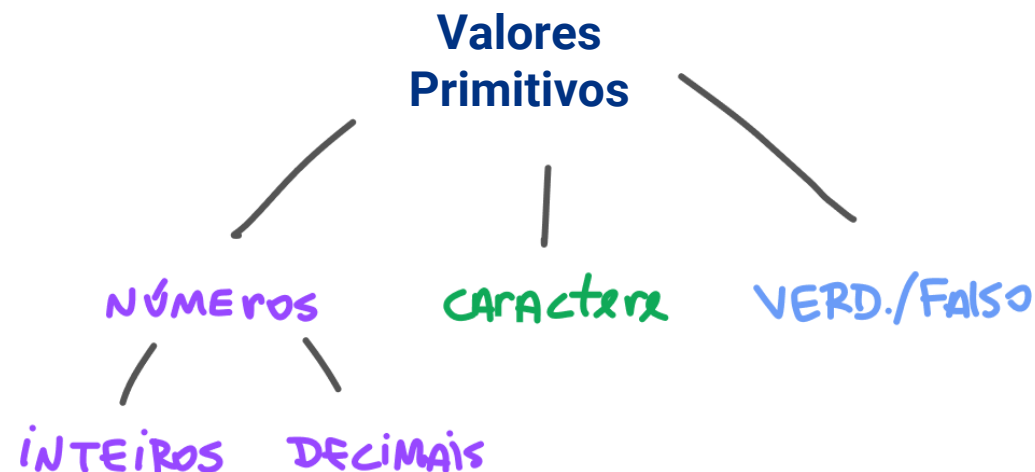
```
>> // Representar sua idade
>> 32
>>
>> // Representar os graus do polo norte
>> -10
>>
>> // Representar sua altura
>> 1.81
>>
>> // Representar 1 ponto e meio a menos
>> -1.5
>>
>> // Representar seu nome
>> "Bruno"
>>
>> // Representar nota que tirava na escola
>> 'B'
>>
>> // Representar a hora atual
>> new Date()
>>
>> // Representar o dia do seu nascimento
>> new Date(1989, 9, 22)
>>
>> // Representar algo verdadeiro
>> true
>>
>> // Representar algo falso
>> false
>>
```



Valores Primitivos

Valores primitivos são aqueles que não podem ser decompostos em unidades menores, eles representam o grau mais elementar que se pode ter em uma linguagem de programação. **A partir deles todas as outras representações são criadas.**

Se mudarmos nosso pensamento ao nível físico da máquina, veríamos que a representação primária sublime é o **dígito binário**, o qual é utilizado pelo processador para realizar suas computações. No entanto, essa seria uma representação pouco significativa para a linguagem, já que na construção de softwares corporativos, a manipulação com números, textos e verificações lógicas predominam.



Dá pra decompor mais?

Dígito binário: 0 ou 1

Agrupamento de 8bits

0101 1100 ⇒ 1byte



Valores Compostos

Valores compostos são aqueles que podem ser decompostos em unidades menores.

Eles são criados a partir de outros valores.

Podemos criar nossos próprios valores compostos, mas qualquer tentativa de criar um valor primitivo, resultaria em um valor composto.

Valores
Compostos

TEXTO {B, R, U, N, O}

DATA E HORA {22, 10, 1989, 22, 40, 10}

Objetos {Bruno, 32, Programador}

Estruturas Recursivas {0, 1, 2, 3, 4, 5}



Parte 02

Funções



Roteiro de Treino

Objetivos

- Conhecer os operadores aritméticos;
- Conhecer os operadores relacionais;
- Conhecer os operadores lógicos;
- Conhecer algumas funções high-level;
- **Apontar as funções primitivas e compostas**

```
>> // Representar a função de soma
>> 10 + 5
>>
>> // Representar a função de multiplicação
>> 10 * 5
>>
>> // Representar a função de divisão
>> 10 / 5
>>
>> // Representar a função de comparação
>> 10 > 5
>> 10 < 10
>> 10 <= 10
>> 10 >= 10
>>
>> // Representar a função de igualdade/desigualdade
>> 10 == 5
>> 10 != 5
>> "Bruno" == "bruno"
>> "Bruno" != "bruno"
>> "Bruno".equalsIgnoreCase("bruno")
>>
>> // Representar uma comparação combinada
>> 10 > 5 && 20 > 30
>> 10 > 5 || 20 > 30
>> 10 >= 0 && 10 <= 10
>>
>> // Representar a função de concatenação
>> "bruno é " + "top! nota " + 10
>>
>> // Representar a função de potência
>> Math.pow(2, 4)
>>
>> // Representar a função de raiz quadrada
>> Math.sqrt(25)
>>
```



Objetivos



Roteiro de Treino

Objetivos

- Perceber os tipos de informação que cada grupo de operador manipula;
- Perceber o tipo de resposta de cada grupo de operador;
- Entender os erros que podem acontecer ao enviar valores incompatíveis para os operadores;

```
>>
>>
>>
>>  /*
>>   *
>>   * Operadores relacionais manipulam apenas números,
>>   * com exceção dos operadores de [des]igualdade que
>>   * também manipulam texto
>>   *
>>   * e retornam booleano
>>   *
>>   */
>>  10 > 5
>>  10 < 5.5
>>  10.5 == 10.5
>>  "bruno" == "bruno"
>>  "bruno" != "Bruno"
>>
>>  // erro
>>  "bruno" > "bruno"
>>  "bruno" > 10.5
>>  true > true
>>
>>
>>
>>
>>
>>
>>
>>
```



Roteiro de Treino

Objetivos

- Perceber os tipos de informação que cada grupo de operador manipula;
- Perceber o tipo de resposta de cada grupo de operador;
- Entender os erros que podem acontecer ao enviar valores incompatíveis para os operadores;

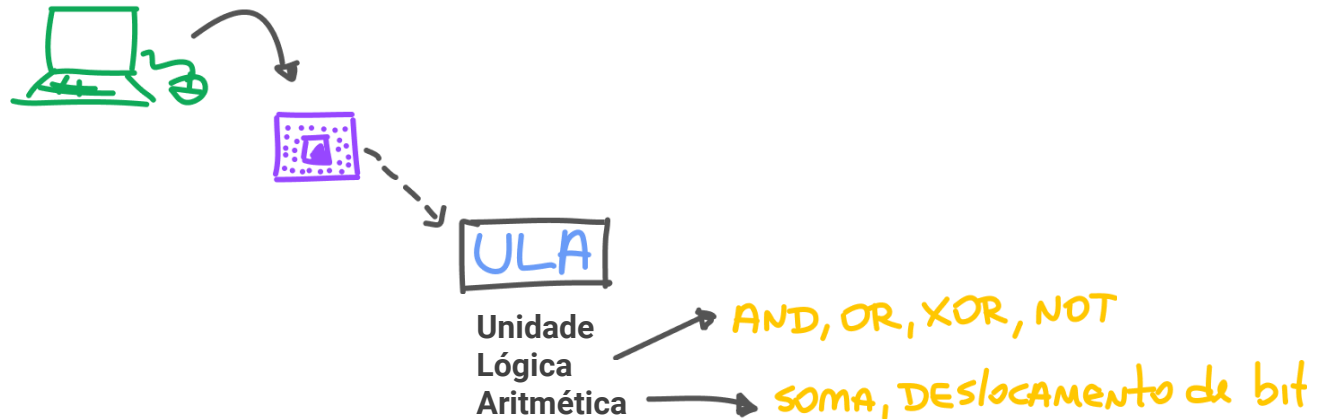
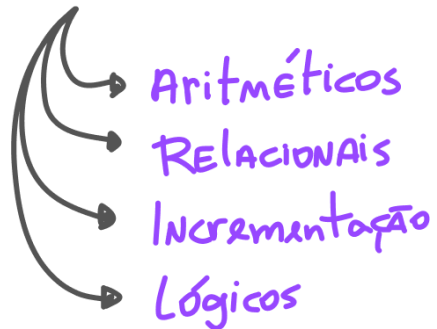
```
>>
>>
>>
>>  /*
>>   *
>>   * Operadores lógicos manipulam apenas
>>   * valores booleanos
>>   *
>>   * e retornam booleano
>>   *
>>   */
>>
>> false && false
>> false && true
>> true  && false
>> true  && true
>>
>> false || false
>> false || true
>> true  || false
>> true  || true
>>
>>
>> 10 > 5 && true
>> 10 > 5 && 5 < 10
>>
>> "bruno" == "bruno" || true
>> "BRUNO".equalsIgnoreCase("bruno") || 5 < 10
>>
>>
>>
>>
```



Funções Primitivas (Operadores)

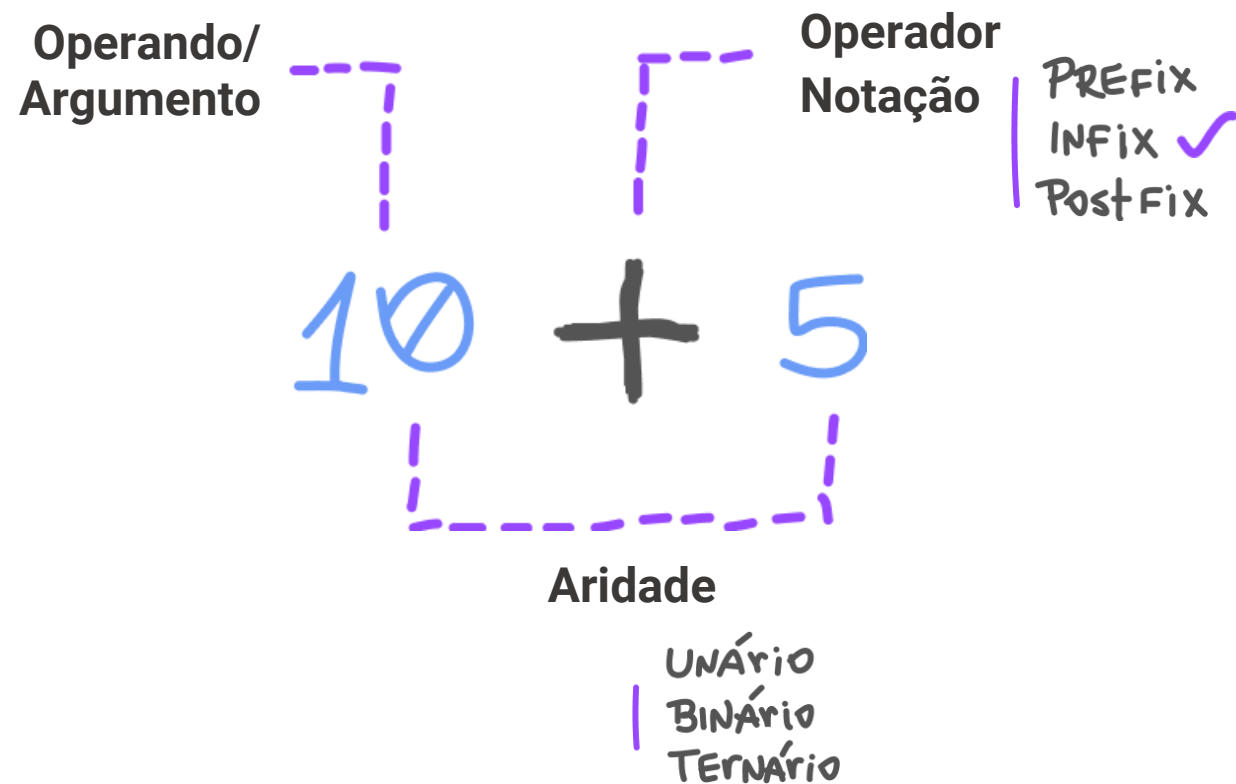
Na linguagem de programação, os cálculos são feitos por funções, que também possuem sua base primitiva, chamados de **Operadores**: *Matemáticos, Incrementação, Relacionais e Lógicos*.

Operadores





Anatomia de um Operador





Operadores Matemáticos

Operações matemáticas envolvem números em seus operandos e em sua resposta. É possível realizar operações com valores fixos ou com variáveis. Quando a operação envolve valores de tipos diferentes, a resposta **será sempre do tipo com o maior conjunto de valores**, ou seja, do conjunto que contém o outro. Também é possível realizar expressões matemáticas contendo mais de uma operação. **A ordem da execução respeita as regras da matemática**, então se quisermos dar prioridade a uma adição ao invés de multiplicação, envolve-lá entre parênteses.

Símbolo	Nome	Exemplo
$\underline{\quad} + \underline{\quad}$	Adição	$10 + 5$
$\underline{\quad} - \underline{\quad}$	Subtração	$10 - 5$
$\underline{\quad} * \underline{\quad}$	Multiplicação	$10 * 5$
$\underline{\quad} / \underline{\quad}$	Divisão	$10 / 5$
$\underline{\quad} \% \underline{\quad}$	Módulo (Resto da divisão)	$10 \% 2$
$\underline{\quad} += \underline{\quad}$	Incrementação por Adição	$x += 5$
$\underline{\quad} -= \underline{\quad}$	Decrementação por Subtração	$x -= 5$
$\underline{\quad} *= \underline{\quad}$	Incrementação por Multiplicação	$x *= 5$
$\underline{\quad} /= \underline{\quad}$	Decrementação por Divisão	$x /= 5$
$\underline{\quad} ++ \underline{\quad}$	Incrementação Pré-Fixado	$++x$
$\underline{\quad} -- \underline{\quad}$	Decrementação Pré-Fixado	$--x$
$\underline{\quad} ++$	Incrementação Pós-Fixado	$x++$
$\underline{\quad} --$	Decrementação Pós-Fixado	$x--$



Operadores Relacionais

Operações relacionais podem envolver qualquer tipo de valores em seus operandos diferente dos operadores matemáticos. Sua especificidade se dá em sempre **retornarem um valor booleano**, ou seja, a ideia de relacionar está intimamente ligada a comparar.

Comparamos se algo é maior, menor, igual, diferente, entre outras disponíveis na linguagem. **Algumas comparações podem não ser implementadas para alguns tipos**, como por exemplo: Não é possível verificar se um texto é maior ou menor que outro.

Símbolo	Nome	Exemplo
$_ > _$	Maior que	$10 > 5$
$_ < _$	Menor que	$10 < 5$
$_ \geq _$	Maior ou igual que	$10 \geq 5$
$_ \leq _$	Menor ou igual que	$10 \leq 5$
$_ == _$	Igual a	$10 == 5$
$_ != _$	Diferente de	$10 != 5$






Operadores Lógicos

Operadores lógicos assim como os relacionais retornam um valor booleano. **Sua característica principal é receber em seus operandos apenas expressões booleanas.** Assim, operadores lógicos trabalham apenas com valores booleanos.

A operação lógica E retornará verdadeiro apenas se seus dois operandos forem verdadeiros, caso contrário retornará falso.

A operação lógica OU retornará verdadeiro se qualquer um de seus dois parâmetros for verdadeiro. Retornará falso apenas se os dois forem falsos.

Símbolo	Nome	Exemplo
	E lógico	10 > 5 && 5 > 0
	OU Lógico	10 < 5 5 > 6
	Negação	!true



Outros Operadores

Símbolo	Nome	Observação	Exemplo
$_ + _$	Concatenação	Junta um texto com outra informação	"Meu nome é: " + " Bruno "
$_ [_]$	Indexação	Acessa uma posição específica de um array	numeros[0]



Funções Compostas (Função)

Na linguagem de programação, designamos funções compostas àquelas criadas a partir de outras funções.

Elas podem ser criadas **pelo programador**, **disponibilizadas pela linguagem**, **libs** ou **frameworks**.

`calcularMedia(6.5, 7, 8.5)` → CRIADAS
Pelo Dev

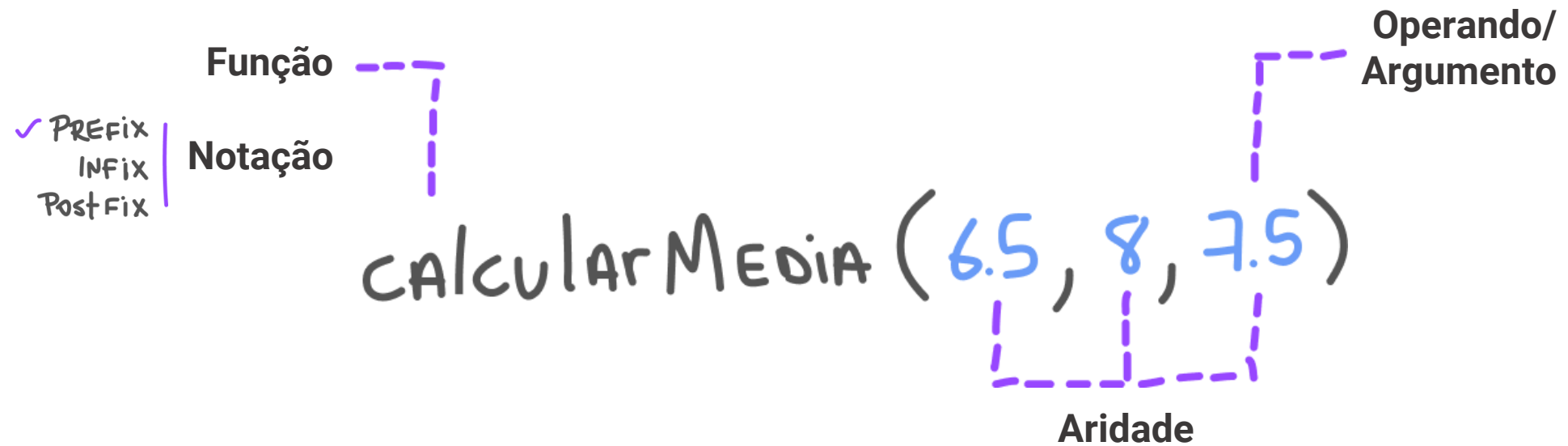
`Math.pow(2, 4)` → DISPONIBILIZADAS
Pela linguagem

`Workbook.getWorkbook(fileLocation)` → de Libs

`db.findAll()` → de Frameworks



Anatomia de uma Função





Funções Matemáticas

Função	Observação	Retorno	Exemplo
<code>abs(_)</code>	Retorna o valor absolute de um número	int/double	<code>double x = Math.abs(-10);</code> // 10
<code>ceil(_)</code>	Retorna o número arredondado para cima	double	<code>double x = Math.ceil(9.1);</code> // 10
<code>floor(_)</code>	Retorna o número arredondado para baixo	double	<code>double x = Math.floor(9.9);</code> // 9
<code>pow(_,_)</code>	Retorna a potencia de um número	double	<code>double x = Math.pow(2, 4);</code> // 16
<code>log10(_)</code>	Retorna o logaritmo de um número na base 10	double	<code>double x = Math.log10(10);</code> // 1
<code>random()</code>	Retorna um valor aleatório entre 0 e 1	double	<code>double x = Math.random();</code> // ?
<code>round(_)</code>	Retorna o valor arredondado de um número	int/long	<code>long x = Math.round(5.6);</code> // 6
<code>sqrt(_)</code>	Retorna a raiz quadrado de um número	Double	<code>double x = Math.sqrt(25);</code> // 5



Funções de Texto

Função	Observação	Retorno	Exemplo
			<pre>string s = "Dev";</pre>
<code>charAt(_)</code>	Retorna o caractere de uma posição	char	<pre>char x = s.charAt(0); // 'D'</pre>
<code>codePointAt(_)</code>	Retorna o Código UNICODE de uma posição	int	<pre>int x = s.codePointAt(0); // 68</pre>
<code>contains(_)</code>	Verifica se um texto existe	boolean	<pre>boolean x = s.contains("v"); // true</pre>
<code>equals(_)</code>	Verifica se é igual a uma string	boolean	<pre>boolean x = s.equals("Dev"); // true</pre>
<code>indexOf(_)</code>	Retorna a posição de um texto	int	<pre>int x = s.indexOf("v"); // 2</pre>
<code>length()</code>	Retorna a quantidade de caracteres	int	<pre>int x = s.length(); // 3</pre>
<code>matches(_)</code>	Retorna se uma expressão regular é aceita	boolean	<pre>boolean x = s.matches("D.v"); // true</pre>
<code>replace(_,_)</code>	Substitui um texto por outro	String	<pre>String x = s.replace("e", "i"); // Div</pre>
<code>substring(_,_)</code>	Recorta uma string	String	<pre>String x = s.substring(1,3); // ev</pre>
<code>toLowerCase()</code>	Retorna todos caracteres em minúsculo	String	<pre>String x = s.toLowerCase(); // dev</pre>
<code>toUpperCase()</code>	Retorna todos caracteres em maiúsculo	String	<pre>String x = s.toUpperCase(); // DEV</pre>
<code>trim()</code>	Remove os espaços do começo e fim	String	<pre>String x = s.trim(); // Dev</pre>



ENCONTRO 04

Significando Valores com Variáveis



Agenda

- Conjuntos
 - Tipos
 - Armazenamento
- Variáveis
 - Tipagem
 - Inferência
 - Casting
 - Coercion
 - Conversões



Parte 01

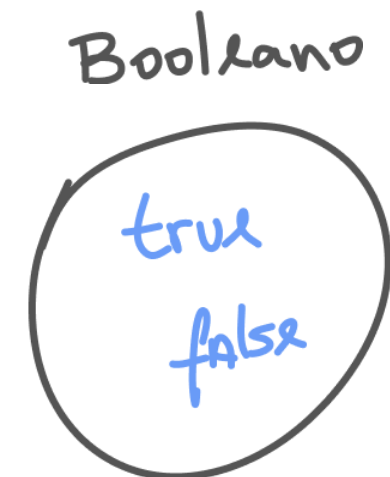
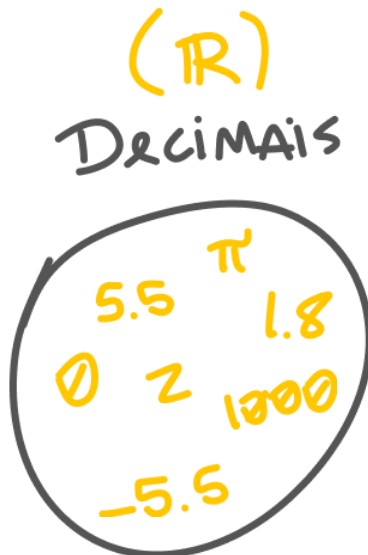
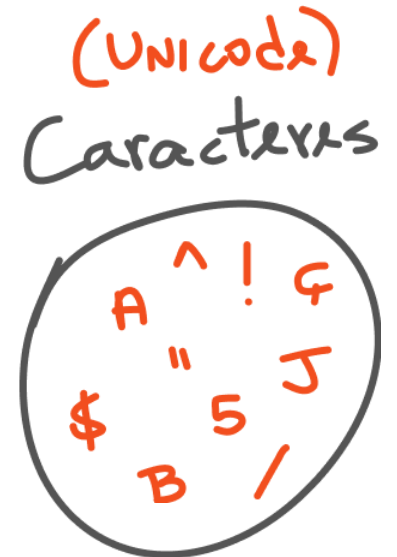
Conjuntos



Tipos de Valores

Um valor está sempre vinculado à um tipo. O tipo determina um **conjunto de valores que possuem mesma representação, e neles, pode ser realizadas as mesmas ações**. Os Tipos referentes aos valores primários são os conjuntos: *números inteiros*, *números reais*, *caracteres* e *booleano*.

Cada tipo ocupa espaço na storages (memória RAM) e possui uma **quantidade finita** de componentes.





Conjuntos primitivos nas Linguagens

Conjunto	Java	CSharp	JavaScript	Python
Inteiro	short, int, long	short, int, long, ushort, uint, ulong	number, bigInt	int
Real	float, double	float, double, decimal	number	float
Caractere	char	char	string	str
Booleano	boolean	bool	boolean	bool



Espaço e Limite

Tipo	Tamanho	Limite
short	2 bytes	-32.768 a 32.767
int	4 bytes	-2.147.483.648 a 2.147.483.647
long	8 bytes	-9.223.372.036.854.775.808 a -9.223.372.036.854.775.807
float	4 bytes	$-3.40282347 \times 10^{38}$ a $1.40239846 \times 10^{-45}$
double	8 bytes	$1.76769313486231570 \times 10^{308}$ a $4.94065645841246544 \times 10^{-324}$
char	2 bytes	Todos caracteres Unicode.
boolean	1 byte	true e false



Conjuntos compostos nas Linguagens

Conjunto	Java	Csharp	JavaScript	Python
Textos	String	string	string	str
Data e Hora	Date	DateTime	object	datetime
Objeto	Object	object	object	-
Lista	ArrayList	List	object	list



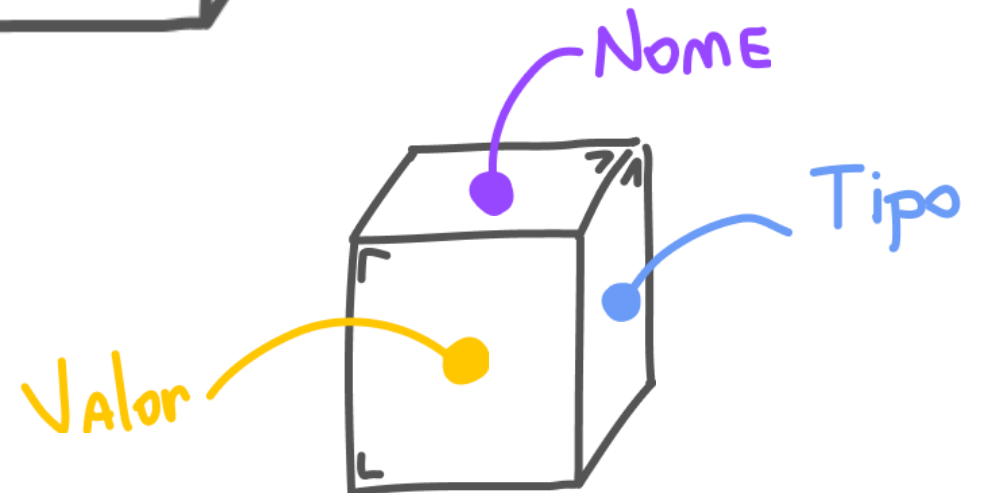
Parte 02

Variáveis



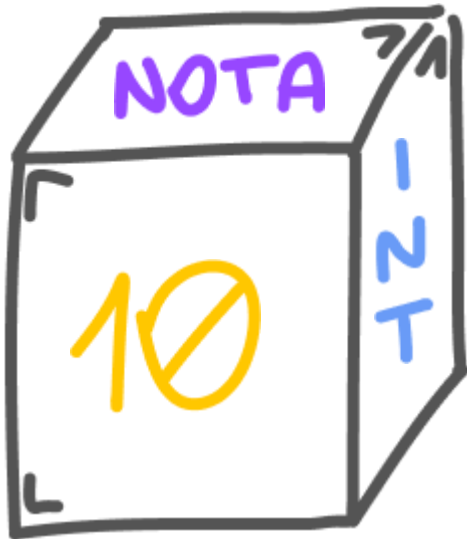
Significando Valores

Qual significado de um valor? **Nenhum**. A única coisa que podemos predicar de um valor é seu tipo. Para dar significado a um valor atribuímos a ele um **Nome**. No entanto, esse nome possui sentido apenas para o programador, a máquina não compreende sua contextualização, como já dissemos, ela apenas computa valores. Na programação, quando nomeamos um valor, estamos criando uma **variável**.





Declarando uma Variável



```
int nota = 10;
```

tipo

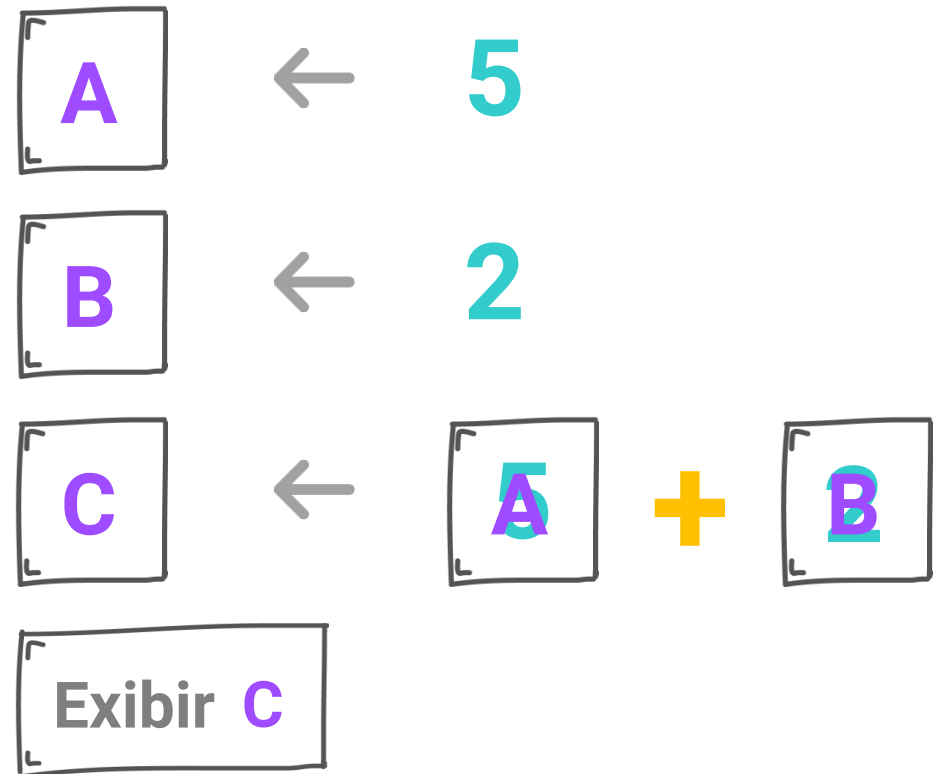
nome

valor



Primeiro Algoritmo

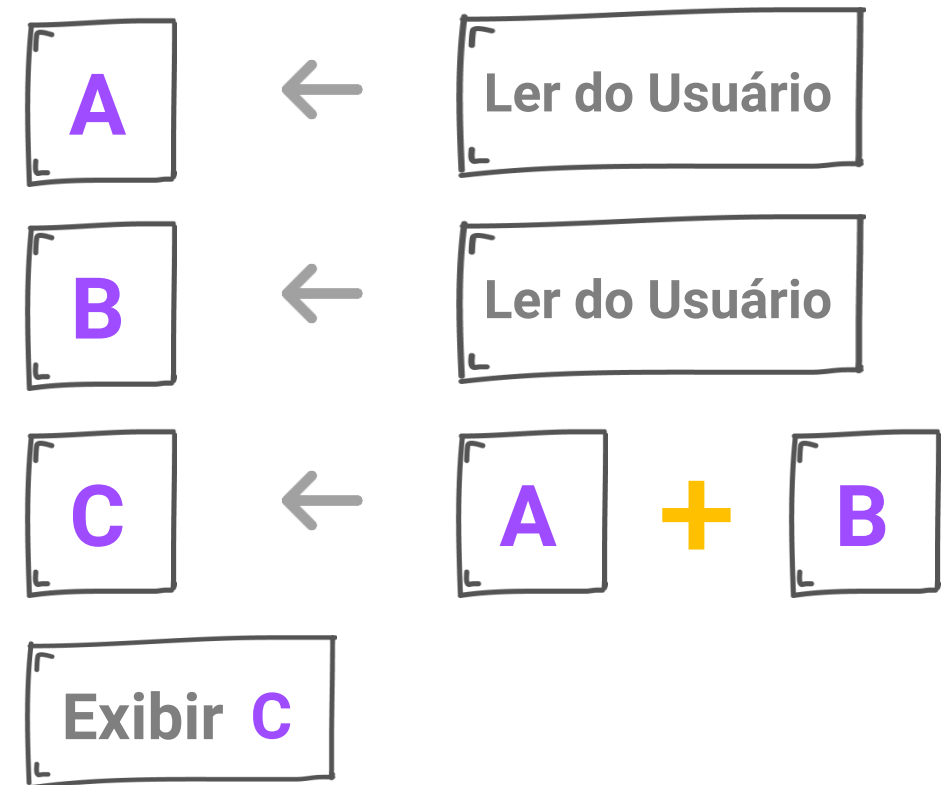
```
>>  
>> int a = 5;  
>> int b = 2;  
>> int c = a + b;  
>> System.out.println(c);  
>>  
>>
```





Primeiro Algoritmo

```
>> import java.util.Scanner;  
>>  
>> Scanner input = new Scanner(System.in);  
>>  
>> int a = input.nextInt();  
>> int b = input.nextInt();  
>> int c = a + b;  
>>  
>> System.out.println(c);  
>>
```

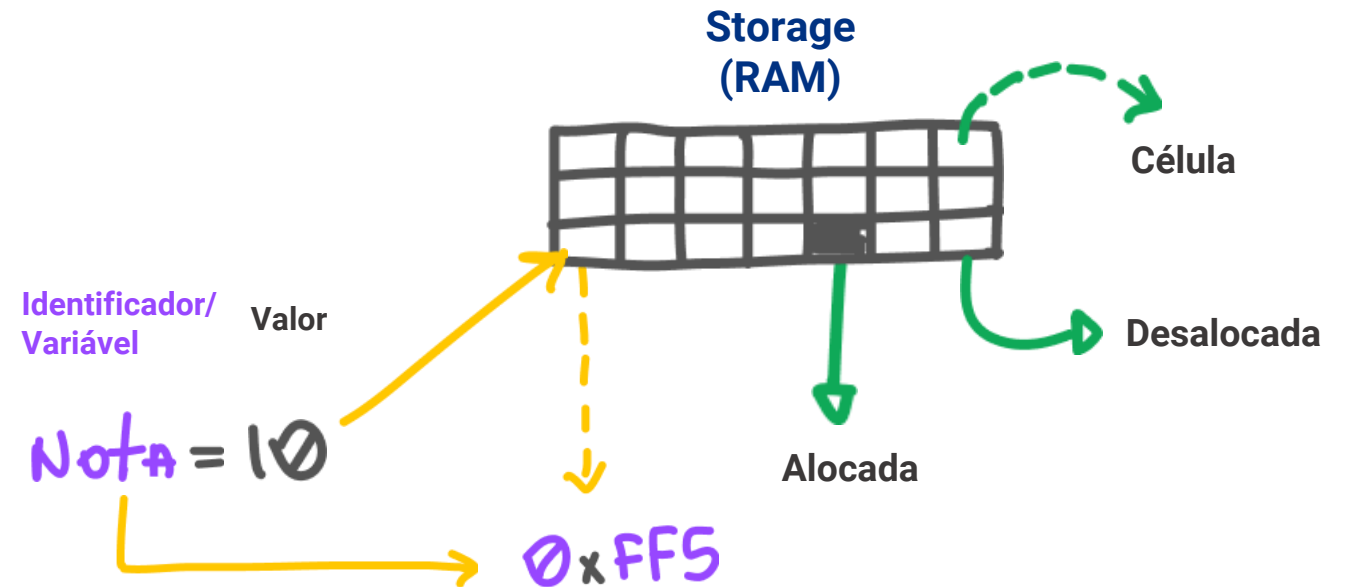




Variáveis e Armazenamento

É possível guardar valores em armazenamentos que chamamos **variáveis**. Possuem esse nome porque podemos **substituir** um valor armazenado por outro. Fisicamente, o armazenamento é a memória RAM, onde cada valor fica guardado em uma das milhões de células disponíveis.

Cada célula possui um endereço físico, mas na linguagem de programação, definimos um **identificador arbitrário** que apontará para esse endereço na memória.





Tipagem da Linguagem

Em linguagens de **tipagem estática** declara-se a variável informando o tipo de valor que ela poderá receber, nesse caso o tipo está conectado a variável e ela nunca poderá mudar de tipo. Já em linguagens de **tipagem dinâmica**, não informa-se o tipo do variável, o tipo não está conectado a variável, podendo ela mudar de tipo dependendo do valor recebido.

Tipagem
Estática

```
>>  
>> int numero = 10;  
>> numero = "Oie"; // erro  
>>
```

Tipagem
Dinâmica

```
>>  
>> let numero = 10;  
>> numero = "Oie"; // ok  
>>
```



Declarando Variáveis em Linguagens Estáticas

```
>>
>>
>>
>> // JAVA
>>
>> // Criando
>>
>>     short x1 = 10;
>>         int x2 = 10;
>>         long x3 = 10;
>>
>>     float x4 = 10.4f;
>>     double x5 = 10.4;
>>
>>     boolean x6 = true;
>>
>>     char x7 = 'A';
>>     String x8 = "Oie";
>>
>>
>> // Alterando
>>
>>     x2 = 15;
>>     x5 = 10.5;
>>     x6 = false;
>>     x7 = 'B';
>>     x8 = "Xau";
>>
>>
>>
>>
```

```
>>
>>
>>
>> // C#
>>
>> // Criando
>>
>>     short x1 = 10;
>>         int x2 = 10;
>>         long x3 = 10;
>>
>>     float x4 = 10.4f;
>>     double x5 = 10.4;
>>
>>     bool x6 = true;
>>
>>     char x7 = 'A';
>>     string x8 = "Oie";
>>
>>     ushort x9 = 10;
>>     uint x10 = 10;
>>     ulong x11 = 10;
>>
>>     decimal x12 = 10m;
>>
>> // Alterando
>>
>>     x2 = 15;
>>     x5 = 10.5;
>>     x6 = false;
>>     x7 = 'B';
>>     x8 = "Xau";
>>
>>
>>
>>
```



Declarando Variáveis em Linguagens Dinâmicas



Roteiro de Treino

em linguagens estáticas – Java

Objetivos

- Concretizar a ideia de significar valores;
- Criar variáveis;
- Alterar variáveis;
- Perceber a necessidade de informar o tipo em uma linguagem estática;
- Conhecer os principais tipos primitivos e compostos oferecidos pela linguagem.

```
>>
>>
>>
>>
>>
>>
>>
>> // Significando valores
>>
>> int idade = 32;
>>
>> long investimento = 1000000000;
>>
>> double altura = 1.80;
>>
>> String professor = "Bruno";
>>
>> char notaBimestre = 'F';
>>
>> boolean temCabelo = false;
>>
>> Date nascimento = new Date(1989, 10, 22);
>>
>>
>>
>>
>>
```



Roteiro de Treino

em linguagens estáticas – Java

Objetivos

- Concretizar a ideia de significar valores;
- Criar variáveis;
- Alterar variáveis;
- Perceber a necessidade de informar o tipo em uma linguagem estática;
- Conhecer os principais tipos primitivos e compostos oferecidos pela linguagem.

```
>>
>>
>>
>>
>> // Expressões matemáticas
>> // - manipulam valores numéricos
>> // - retornam valores numéricos
>> // - quando os valores são de tipos diferentes,
>> permanece o tipo maior
>>
>> int a1 = 10 + 5;
>> int a2 = 10 + 5 + 3 / 3;
>> int a3 = (10 + 5 + 3) / 3;
>> int a4 = ((2 + 2) * 3 + 3) / 3;
>>
>> double a5 = 1.5 + 1.5;
>> double a6 = 10 + 5;
>> double a7 = a1 + 5.0;
>>
>>
>> // erro
>> int e1 = 10 + 5.0;
>> int e2 = a1 + 5;
>> int e3 = a1 + 5;
>>
>>
>>
>>
>>
```



Roteiro de Treino

em linguagens estáticas – Java

Objetivos

- Concretizar a ideia de significar valores;
- Criar variáveis;
- Alterar variáveis;
- Perceber a necessidade de informar o tipo em uma linguagem estática;
- Conhecer os principais tipos primitivos e compostos oferecidos pela linguagem.

```
>>
>>
>>
>>
>>
>> // Expressões relacionais
>> // - manipulam no geral números
>> //   (==, != manipulam texto também
>> // - retornam booleano
>>
>>
>> int x1 = 10;
>> int x2 = 5;
>>
>>
>> boolean a1 = x1 > 5;
>> boolean a2 = x1 == x2;
>> boolean a5 = "Oie" == "Oie";
>> boolean a6 = "Oie" != "Oie";
>>
>>
>> // erro
>>
>> int e1 = 10 > 5;
>> boolean e2 = "10" > "5";
>>
>>
>>
>>
```



Roteiro de Treino

em linguagens estáticas – Java

Objetivos

- Concretizar a ideia de significar valores;
- Criar variáveis;
- Alterar variáveis;
- Perceber a necessidade de informar o tipo em uma linguagem estática;
- Conhecer os principais tipos primitivos e compostos oferecidos pela linguagem.

```
>>
>> // Expressões lógicas
>> // - manipulam booleano
>> // - retornam booleano
>>
>> int x1 = 10;
>> int x2 = 5;
>> String x3 = "Oie";
>>
>> boolean a1 = true && true;
>> boolean a2 = true && false;
>> boolean a3 = false && true;
>> boolean a4 = false && false;
>>
>> boolean a5 = true || true;
>> boolean a6 = true || false;
>> boolean a7 = false || true;
>> boolean a8 = false || false;
>>
>> boolean a9 = x1 >= 0 && x1 <= 10;
>> boolean a10 = x1 >= x2 && x2 > 0 || x3 == "Oie";
>>
>> // erro
>>
>> boolean e1 = 10 || 5;
>> boolean e2 = "10" && "5";
>> boolean e3 = x1 >= 0 && <= 10;
>> boolean e4 = x1 >= 0 <= 10;
>>
```



Parte 03

Inferência, Cast, Coercion e Conversão entre Tipos



Inferência de Tipo

em linguagens estáticas

É a capacidade da linguagem em **detectar automaticamente o tipo** da variável através de um valor. A inferência de tipo acontece naturalmente nas linguagens de tipagem dinâmica.

Quando na linguagem **um valor está contido em dois tipos diferentes**, um deles é escolhido para ser o 'padrão', no entanto, é possível direcionar a linguagem para que use um tipo específico utilizando **sufixos**.

Tipo	Sufixo
long	L
float	F
double	D

```
>>
>>
>>
>>
>>
>>
>>
>>
>>
// Inferência com tipo Padrão
>> var x1 = 10;
>> var x2 = 10.4;
>> var x3 = true;
>> var x4 = 'B';
>> var x5 = "Bruno";
>>
>>
>>
>>
// Inferência indicada por sufixo
>> var x1 = 10L;
>> var x2 = 10.4F;
>> var x2 = 10.4D;
>> var x2 = 10.4M;
>>
>>
>>
>>
>>
>>
```



em linguagens estáticas

As principais são boxing e unboxing:

- **Boxing** acontece quando um valor é convertido para Object, ou uma interface implementada pelo valor.
- **Unboxing** acontece quando um valor é convertido para seu tipo original.

```
int x1 = 10;
char x2 = 'B';
boolean x3 = true;

// Boxing
Object x4 = (Object)x1;
Object x5 = (Object)x2;
Object x6 = (Object)x3;

// Unboxing
int x7 = (int)x4;
int x8 = (char)x5;
int x9 = (boolean)x6;
```

Coercion

É a **conversão implícita**, automática feita pela linguagem em determinadas situações. Algumas linguagens oferecem coerção em diversas situações, outras, em situações bem restritas.



em linguagens estáticas



ENCONTRO 05

Eficácia do Reuso: Funções e Parâmetros



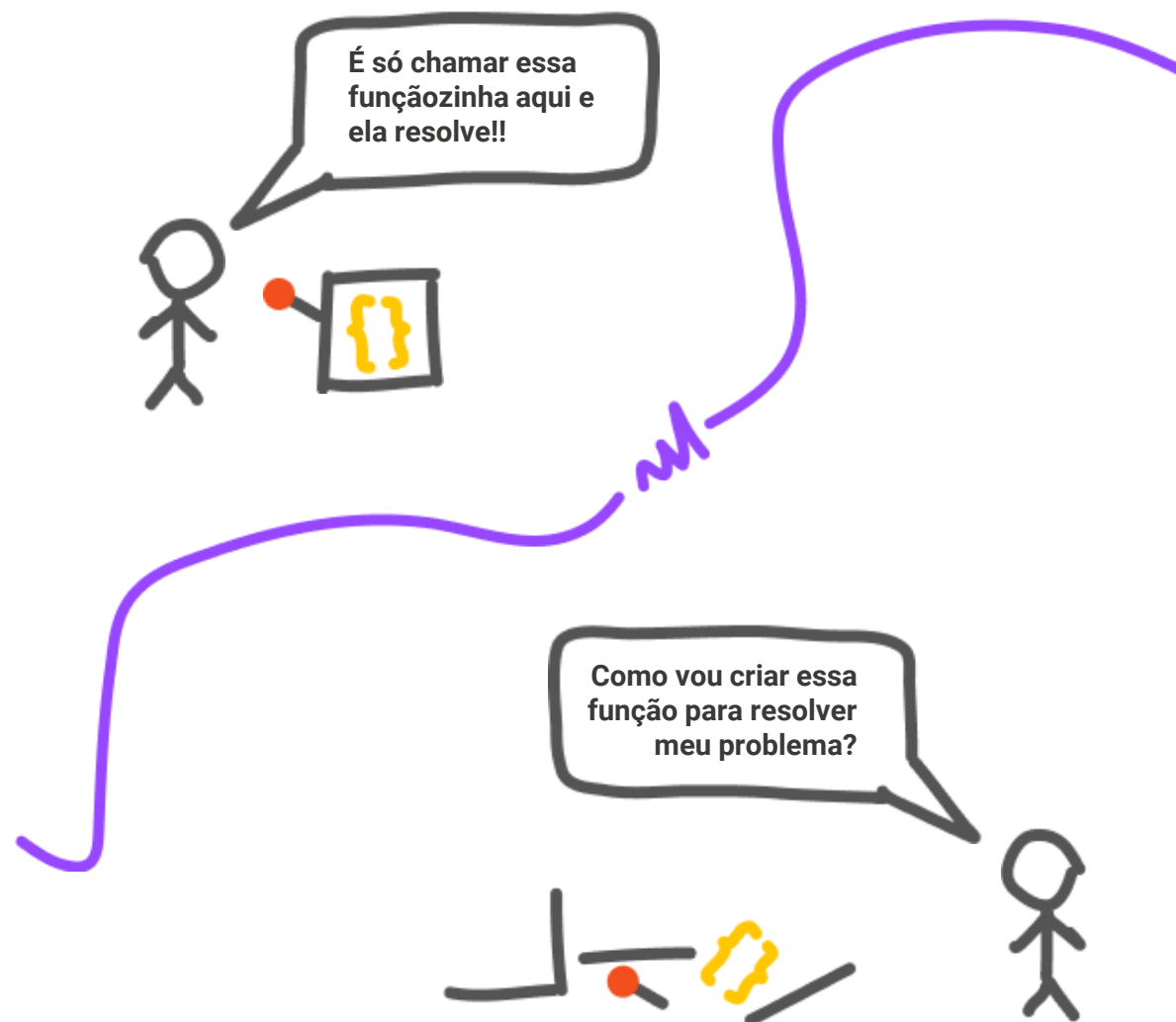
Agenda

- Programador Application vs Implementor
- Princípio do Reuso
- Ciclo de Vida de Variáveis
- Máquina de Função
- Composição de Funções



Programador Application vs Implementor

O mesmo programador assume dois pontos de vistas enquanto programa. Quando sua preocupação está em **usar uma função** já criada previamente, ele assume o ponto de vista **Application**. Já quando sua preocupação está concentrada em **como criar a função** e nos detalhes de sua implementação, ele toma o ponto de vista **Implementor**.



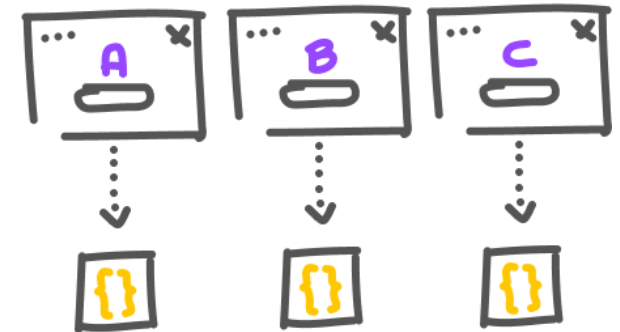


Princípio do Reuso

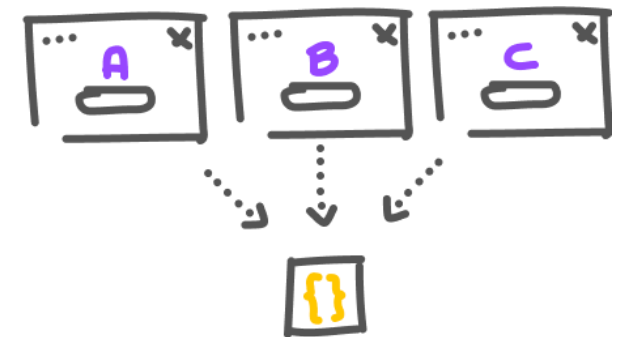
Podemos encontrar em diversos pontos de um sistema, **a mesma ação sendo realizada**. Quais problemas podem ocorrer, se tais ações forem programadas repetidamente para cada ponto do sistema? É certo que o esforço de escrever o mesmo código várias vezes é um problema presente, e se pensarmos no futuro, imagine a necessidade de alterar a regra de uma dessas ações, dezenas de códigos repetidos precisariam ser alterados juntamente, o que seria um custo caro na **manutenção do sistema**.

O **Princípio do Reuso** diz que devemos identificar ações (algoritmos) que podem ser usados em mais de um ponto do sistema e atribuir-lhes um nome para que seu conjunto de instruções possa ser referenciado por tal nome. Na programação, **a Função é responsável por tal tarefa**. A efetividade do princípio do reuso se dá pelo correto uso de **Parâmetros**, os quais representam os valores passíveis de mudança no algoritmo, e que possibilitam o recebimento desses valores no momento de sua execução, **prevenindo assim, a função de executar sempre a mesma coisa**.

Sair desse
mentalidade ;(



Para esse
mentalidade ;)





Roteiro de Treino

em linguagens estáticas

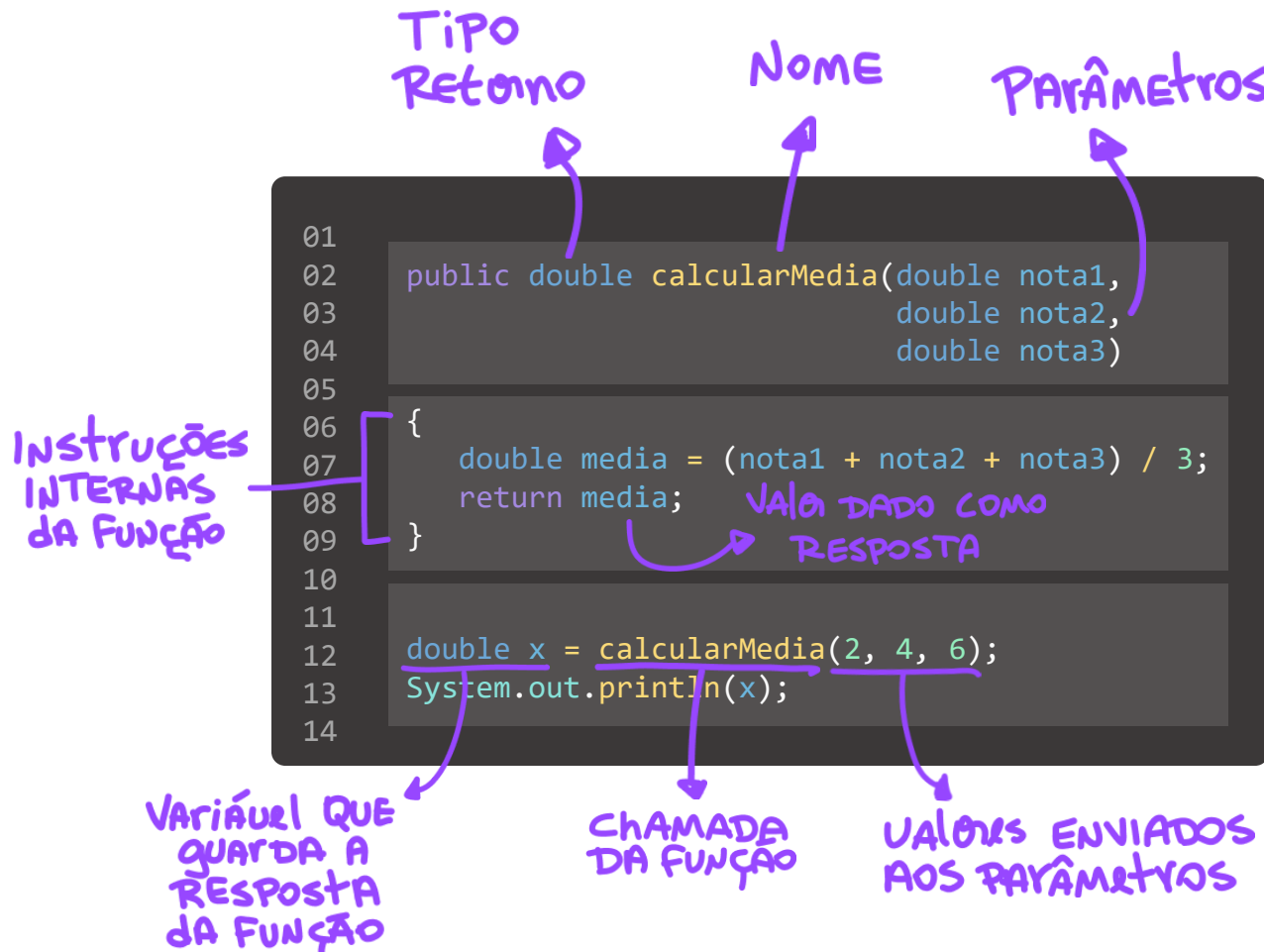
Objetivos

- Entender o algoritmo para cálculo de média.
- Nomear um conjunto de código para trabalhá-lo como se fosse um.
- Identificar as características de Reuso Fraco.
- Aplicar a técnica de parametrização para oferecer Reuso Forte.

```
>>
>> // Um conjunto de instruções para calcular a média
>>
>> double nota1 = 5.0;
>> double nota2 = 7.0;
>> double nota3 = 8.0;
>>
>> double media = (nota1 + nota2 + nota3) / 3;
>>
>>
>> // Nomeando um conjunto de comandos - Reuso Fraco (SEM parâmetro)
>>
>> public double calcularMedia()
>> {
>>     double nota1 = 5.0;
>>     double nota2 = 7.0;
>>     double nota3 = 8.0;
>>
>>     double media = (nota1 + nota2 + nota3) / 3;
>>     return media;
>> }
>>
>> double x = calcularMedia();
>>
>>
>> // Nomeando um conjunto de instruções - Reuso Forte (COM parâmetros)
>>
>> public double calcularMedia(double n1, double n2, double n3)
>> {
>>     double media = (n1 + n2 + n3) / 3;
>>     return media;
>> }
>>
>> double x = calcularMedia(2, 4, 6);
>>
```



Sintaxe de uma Função



Uma função é composta de sua **assinatura** e de seu **corpo de implementação**.

A assinatura é composta de **Nome, Parâmetros e Tipo de Retorno**. Na assinatura estão as informações para que o **programador application** utilize a função.

O corpo é composto das instruções que serão executadas quando a função for chamada. No corpo está o algoritmo criado pelo **programador implementor** quando ele criou a função.

Depois de criada, a função pode ser chamada (executada), para isso devemos **chamá-la pelo nome, enviar os valores aos parâmetros e guardar sua resposta**.



Entendendo Funções

No campo da ciência da computação, uma função (f) possui significado diferente quando comparada à matemática. Na matemática, uma função corresponde a associação dos elementos de dois conjuntos. Na ciência da computação, uma função é um elemento capaz de nomear uma sequência de operações realizadas a partir de valores de entrada com objetivo de chegar a um resultado, ou valor de saída.

Função que dobra um número

$$f(10) = 20$$

Função que soma dois números

$$f(10, 5) = 15$$

Função que calcula a média de três notas

$$f(10, 5, 3) = 6$$



Outros cenários com Funções

Função que calcula a metade de um número

$$f(11) = 5.5$$

Função que calcula total de uma compra a partir do valor total e desconto em %

$$f(1000, 10) = 900$$

Função que calcula a área do quadrado

$$f(10) = 100$$

Função que verifica se a pessoa é de Libra a partir do mês e dia

$$f(\text{"Outubro"}, 22) = \text{true}$$

Função que calcula a área de um triângulo

$$f(10, 5) = 25$$

Função que verifica se uma cor é primária

$$f(\text{"azul"}) = \text{true}$$



Implementando de Funções

Uma função (f) pode ser vista por dois ângulos. O primeiro, vimos anteriormente, onde se envia os valores à função e ela retorna uma resposta. Nessa visão, a preocupação está apenas em **usar a função**. A segunda visão preocupa-se em **como** a função deve ser construída, ou seja, **quais operações** devem ser feitas com os **valores recebidos** para obter-se o **resultado final**.

Função que dobra um número

$$f(a) \Rightarrow a * 2$$

Função que soma dois números

$$f(a, b) \Rightarrow a + b$$

Função que calcula a média de três notas

$$f(a, b, c) \Rightarrow (a + b + c) / 3$$



Outros cenários com Funções

Função que calcula a metade de um número

$$f(a) \Rightarrow a / 2$$

Função que calcula a área de um triângulo

$$f(a, b) \Rightarrow a * b / 2$$

Função que calcula a área do quadrado

$$f(a) \Rightarrow a * a$$

Função que calcula total de uma compra a partir do valor total e desconto em %

$$f(a, b) \Rightarrow a - (a * b / 100)$$



Outros cenários com Funções

Função que verifica se a pessoa é de Libra a partir do mês e dia

```
f(a, b) => (a == "Outubro" && b <= 22) ||  
           (a == "Setembro" && b >= 23)
```

Função que verifica se uma cor é primária

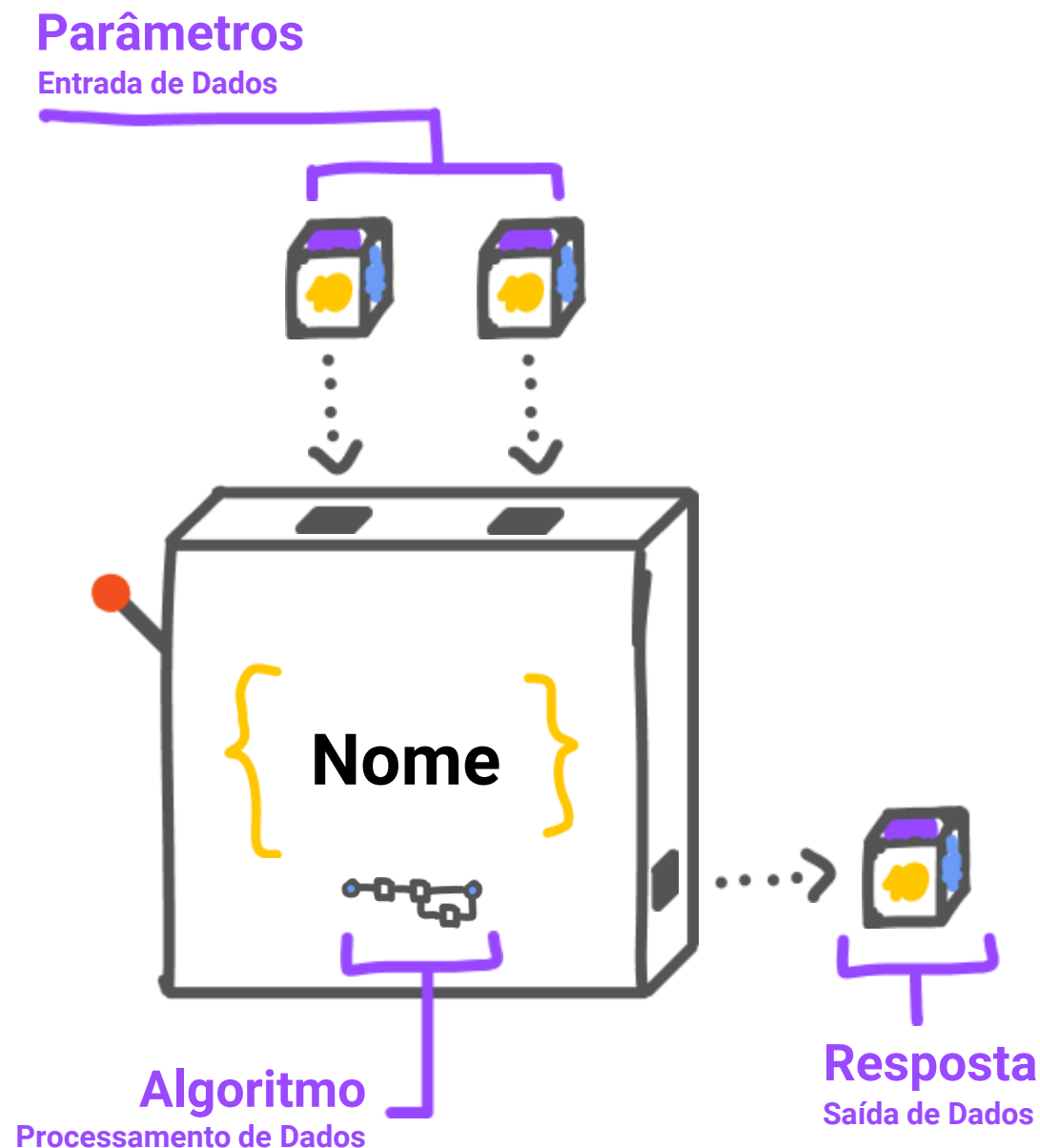
```
f(a) => a == "azul" ||  
        a == "vermelho" ||  
        a == "amarelo"
```



Máquina de Função

Uma função pode ser pensada como uma máquina definida por programador (**Implementor**) e passível de ser chamada a qualquer momento (**Application**). Para chamar uma função precisamos seguir algumas regras, como:

1. Chamá-la pelo **nome** correto considerando caracteres maiúsculos e minúsculos.
2. Enviar os **argumentos** (valores) que ela pede para funcionar corretamente.
3. Armazenar o valor de sua **resposta** para usá-lo posteriormente.





Máquina de Função (CalcularMedia)

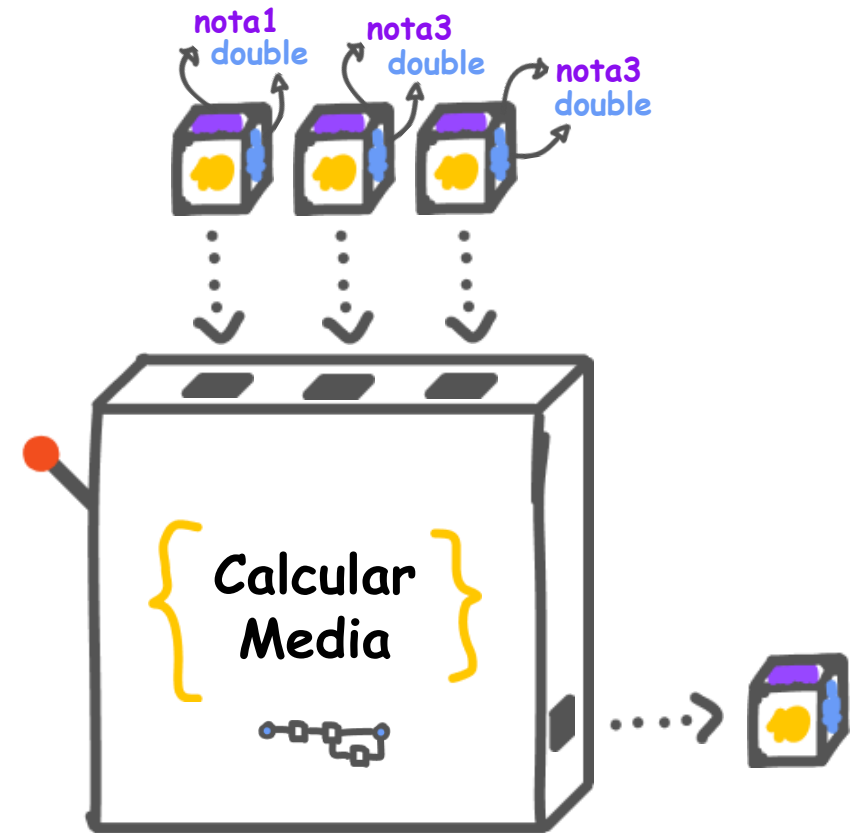
```
01  
02  public double calcularMedia(double nota1,  
03                             double nota2,  
04                             double nota3)  
05  
06  {  
07      double media = (nota1 + nota2 + nota3) / 3;  
08      return media;  
09  }  
10  
11  
12  
13  double x = calcularMedia(2, 4, 6);  
14  System.out.println(x);  
15  
16  
17
```

$$f(a, b, c) \Rightarrow (a + b + c) / 3$$



Máquina de Função (CalcularMedia)

```
01  
02 public double calcularMedia(double nota1,  
03                             double nota2,  
04                             double nota3)  
05  
06 {  
07     double media = (nota1 + nota2 + nota3) / 3;  
08     return media;  
09 }  
10  
11  
12  
13 double x = calcularMedia(2, 4, 6);  
14 System.out.println(x);  
15  
16  
17
```

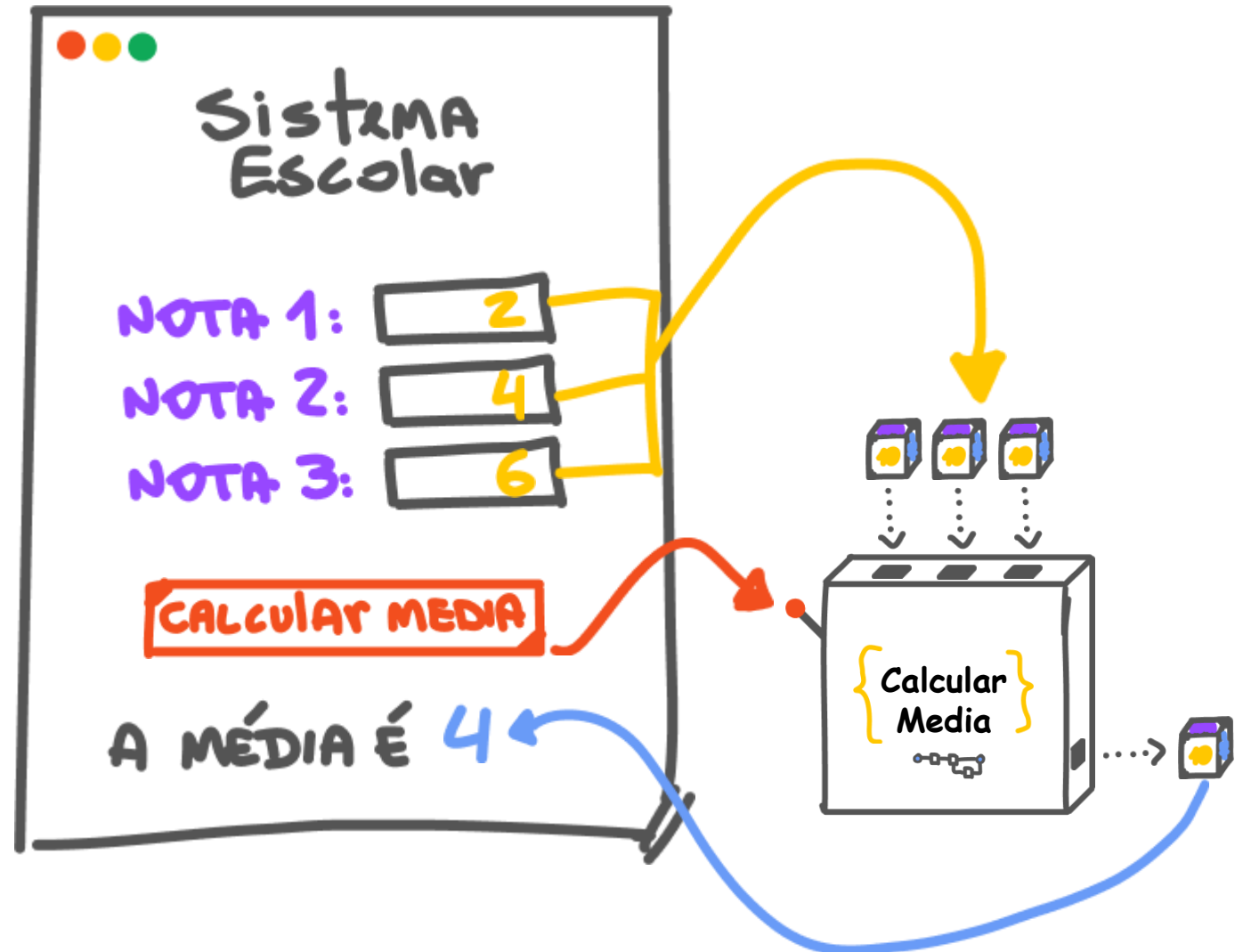




Máquina de Função

vista por uma INTERFACE GRÁFICA

A máquina de função é essencial para entendermos o conceito abstrato de uma função. Um outro ângulo que pode trazer uma visão mais prática sobre a máquina de função é **observá-la como um programa de interface gráfica que o usuário interage**, ou seja, que o usuário envia informações (**entrada de dados**), clica em um botão (**processamento de dados**) e obtém uma resposta (**saída de dados**).




Ciclo de Vida de Variáveis

Na estrutura sintática de uma função podemos encontrar diversas declarações, duas delas são os **parâmetros** e as **variáveis**. Devemos considerar:

1. Tudo que é criado dentro da função é alocado na memória, e ao término da função, desalocado.
2. Nomes/Identificadores criados dentro da função não podem ser acessados fora da função.
3. Nomes/Identificadores iguais, criados em funções diferentes não são a mesma coisa, são como 2 pessoas que possuem o mesmo nome.

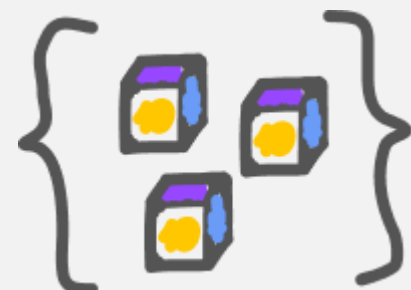
Dobro1



A diagram showing three memory blocks, each containing a yellow circle and a blue arrow, grouped by a large curly brace. This represents the local variables and parameters of the Dobro1 function.

\neq

Dobro2



A diagram showing three memory blocks, each containing a yellow circle and a blue arrow, grouped by a large curly brace. This represents the local variables and parameters of the Dobro2 function. The diagram is identical to the one for Dobro1, illustrating that despite having the same name, the memory spaces are separate.

```
public double Dobro1(double n)
{
    double x = n * 2;
    return x;
}

public double Dobro2(double n)
{
    double x = n * 2;
    return x;
}
```



Resolução de Problemas

Como devemos raciocinar para IMPLEMENTAR e USAR uma Função?

1. Compreender o **problema do enunciado** e resolvê-lo utilizando raciocínios **não computacionais**;
2. Criar a **Máquina de Função**;
 - 2.1. Identificar o **nome** da Função;
 - 2.2. Identificar os **parâmetros** da Função;
 - 2.3. Identificar que **tipo** de informação a Função **retorna**;
3. Implementar o **código na linguagem de programação**:
 - 3.1. A partir da Máquina de Função, criar o código referente a **primeira linha da função (assinatura)**;
 - 3.2. Se perguntar quais operações (*matemáticas, relacionais, lógicas,...*) precisam ser feitas **com os valores de entrada** para chegar na resposta da função;
4. **Chamar a função** criada **enviando os valores** (argumentos) para que ela possa ser executada;
5. **Guardar a resposta** da execução da Função em uma variável;
6. Exibir a resposta no Terminal;

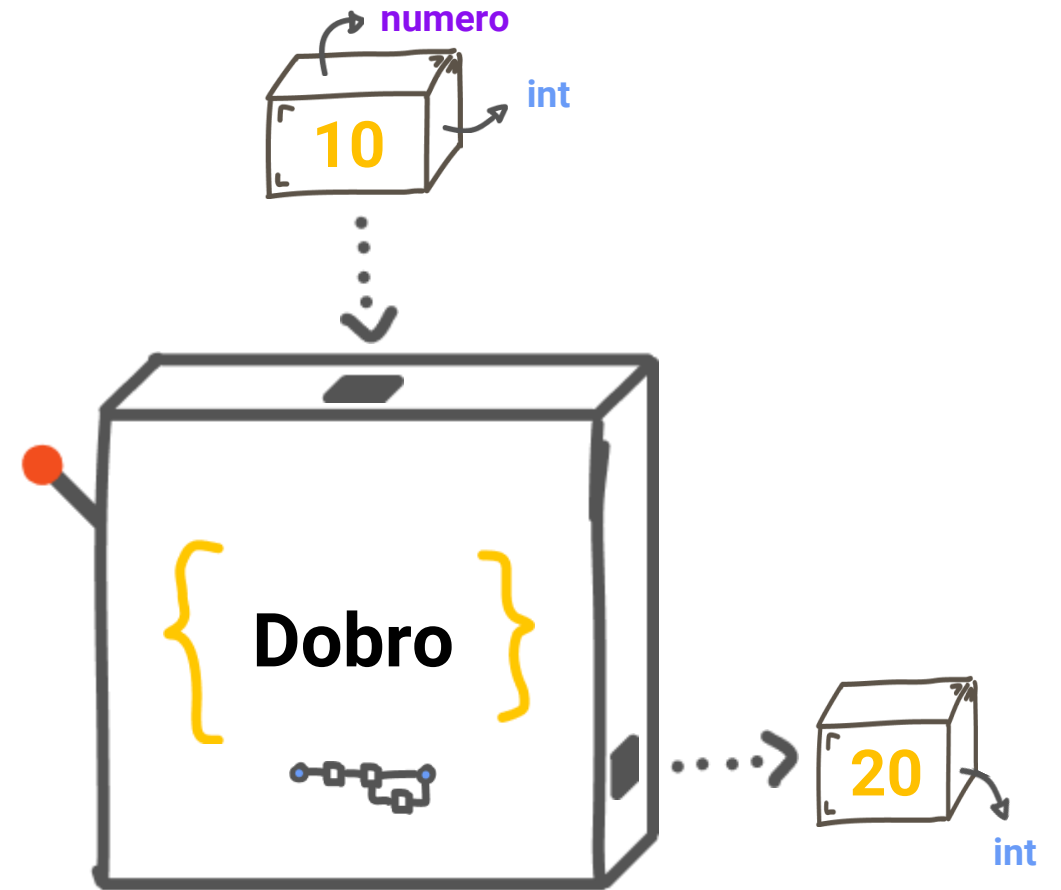




Máquina de Função (Dobro)

em linguagens estáticas

```
>> public int dobro(int numero)
>> {
>>     int d = numero * 2;
>>     return d;
>> }
>>
>> int x = dobro(10);
>> System.out.println(x);
>>
```

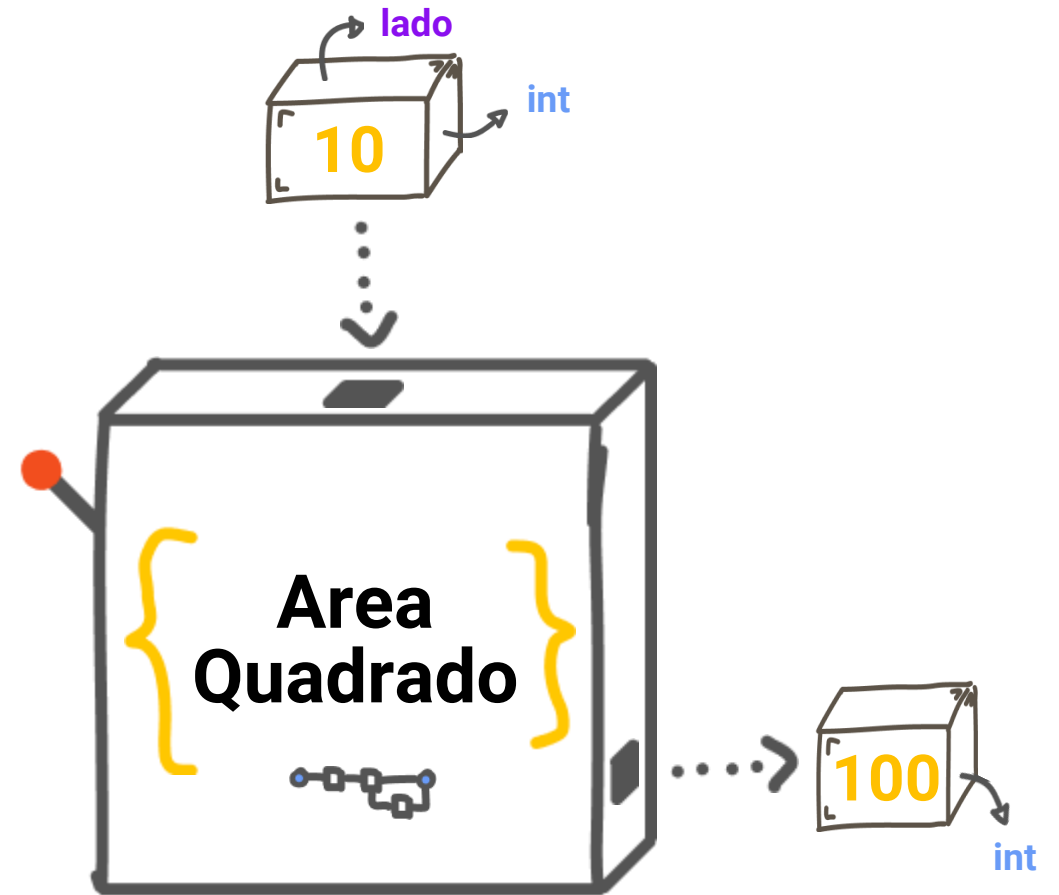




Máquina de Função (AreaQuadrado)

em linguagens estáticas

```
>> public int areaQuadrado(int lado)
>> {
>>     int area = lado * lado;
>>     return area;
>> }
>>
>> int x = areaQuadrado(10);
>> System.out.println(x);
>>
```

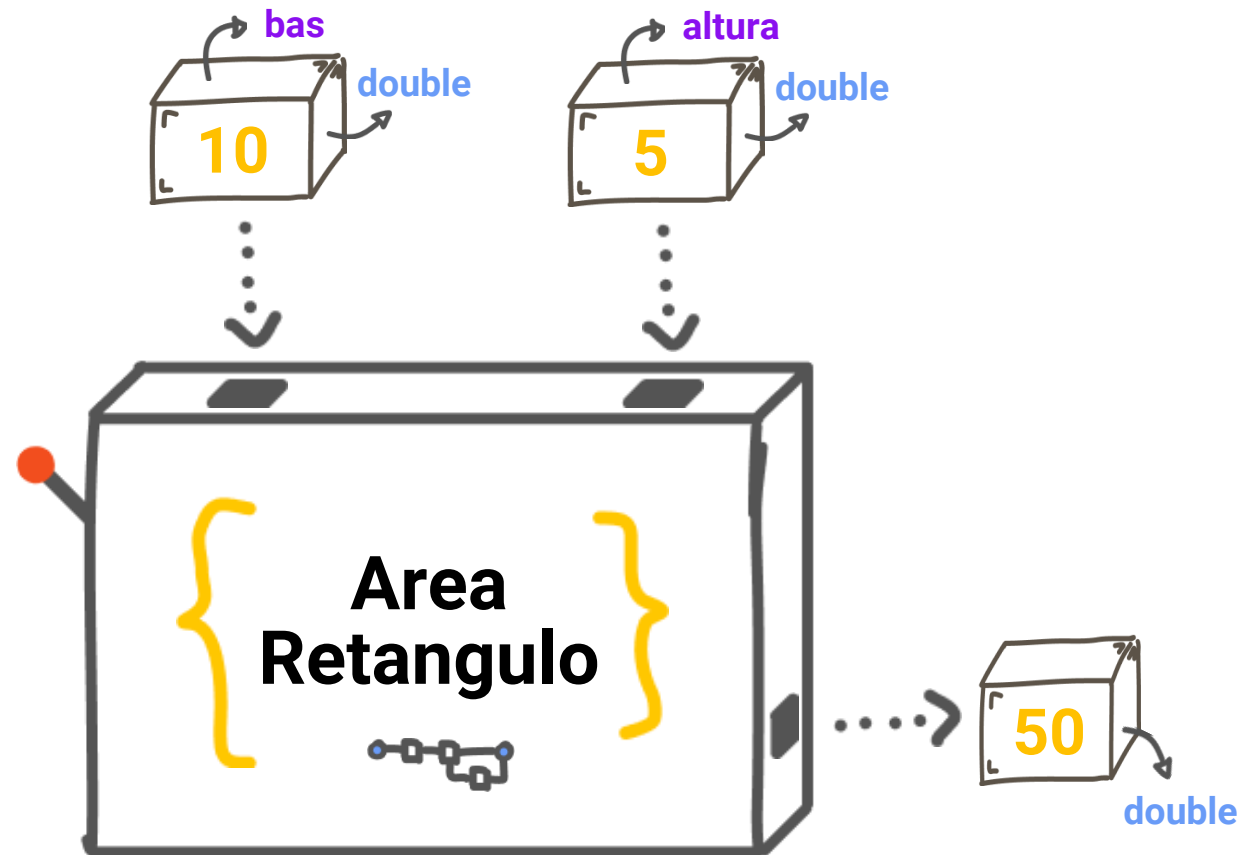




Máquina de Função (AreaRetangulo)

em linguagens estáticas

```
>> public double areaRetangulo(double bas,  
>>                             double altura)  
>> {  
>>     double area = bas * altura;  
>>     return area;  
>> }  
>>  
>> double x = areaRetangulo(10, 5);  
>> System.out.println(x);  
>>
```

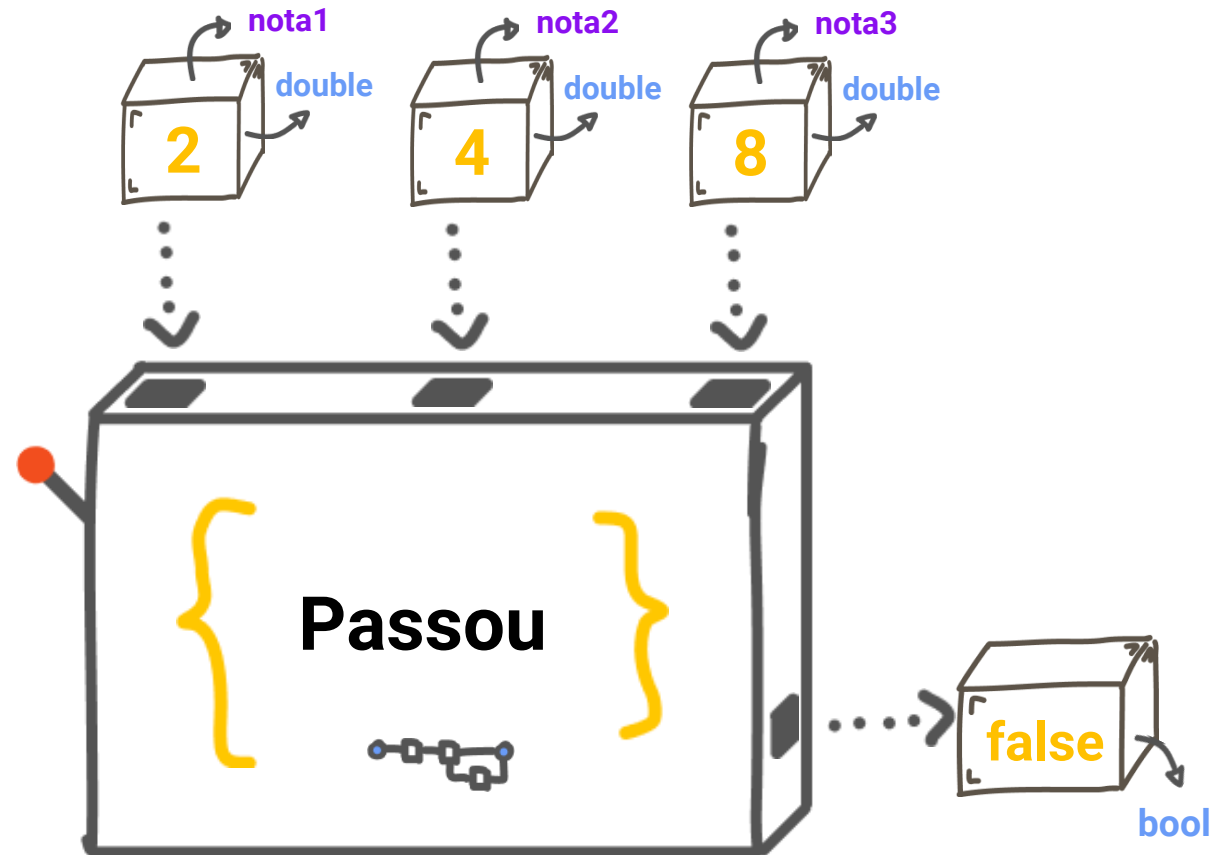




Máquina de Função (Passou)

em linguagens estáticas

```
>> public boolean passou(double nota1,  
>>                        double nota2,  
>>                        double nota3)  
>> {  
>>     double media = (nota1 + nota2 + nota3) / 3;  
>>     boolean p = media >= 5;  
>>     return p;  
>> }  
>>  
>> boolean x = passou(2, 4, 8);  
>> System.out.println(x);
```

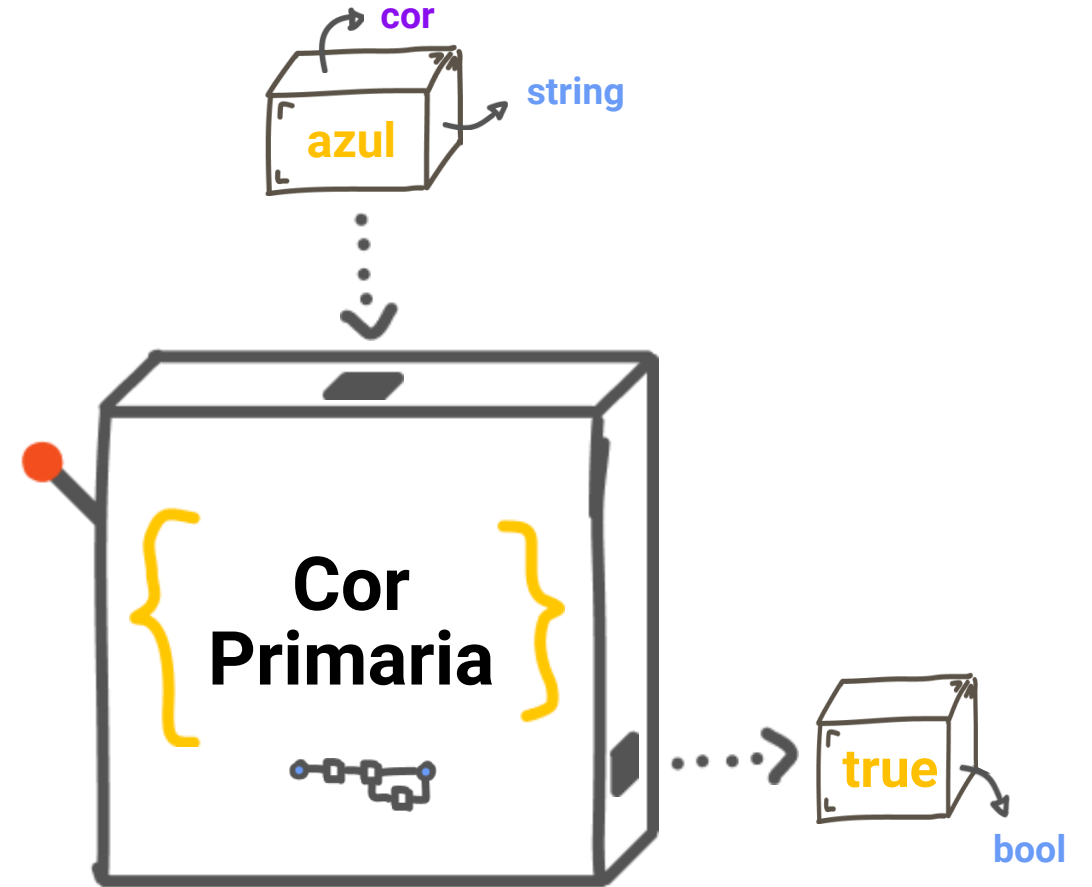




Máquina de Função (É Cor Primária)

em linguagens estáticas

```
>> public boolean corPrimaria(string cor)
>> {
>>     boolean primaria = cor == "azul" ||
>>                         cor == "amarelo" ||
>>                         cor == "vermelho";
>>     return primaria;
>> }
>>
>> boolean x = corPrimaria("azul");
>> System.out.println(x);
>>
```

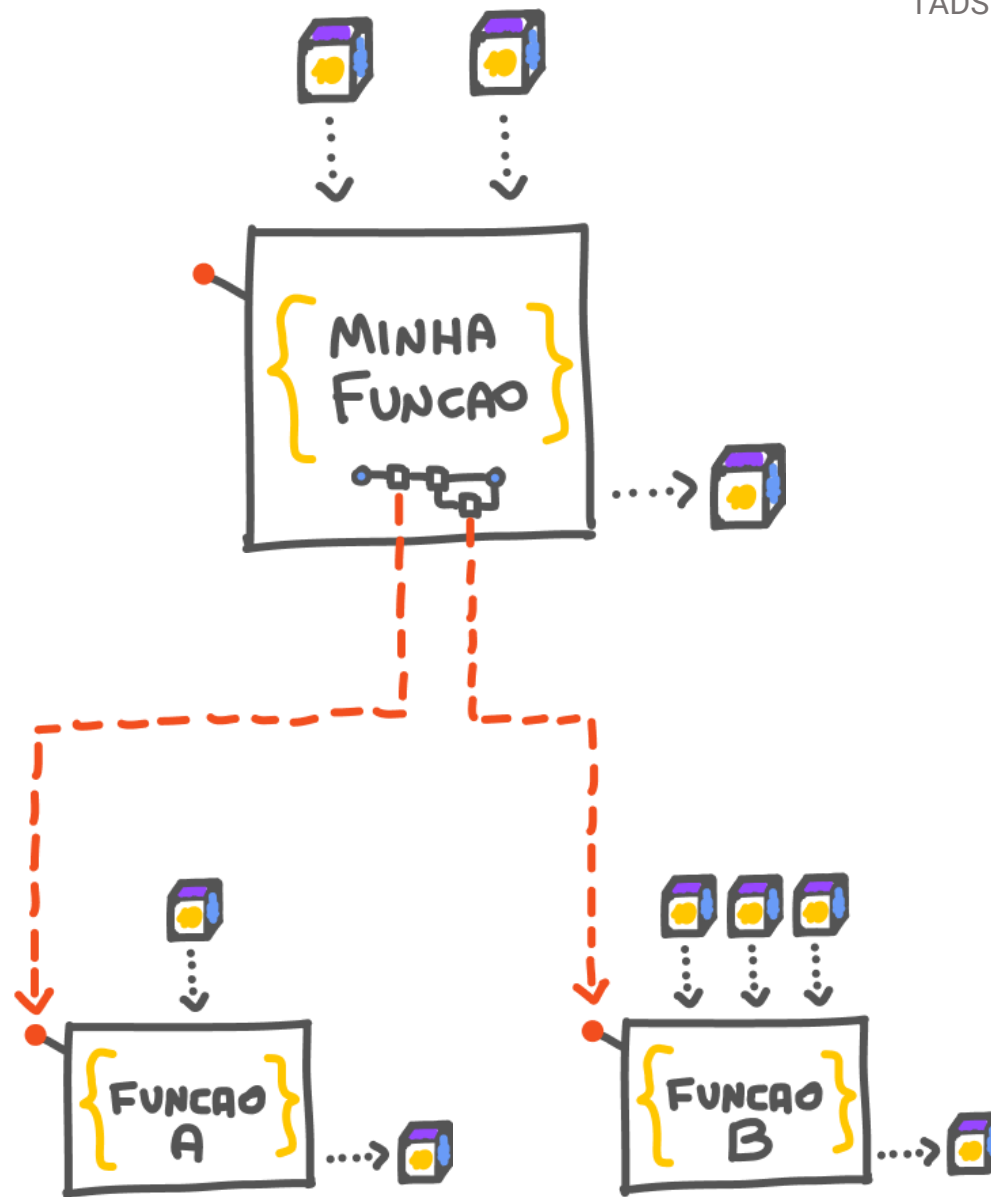




Composição de Funções

Pelo princípio do Reuso estabelecemos a ideia de que ações devem possuir certa **independência** para resolverem problemas próprios, mas também servir como **mini-tarefa** na construção de uma tarefa maior.

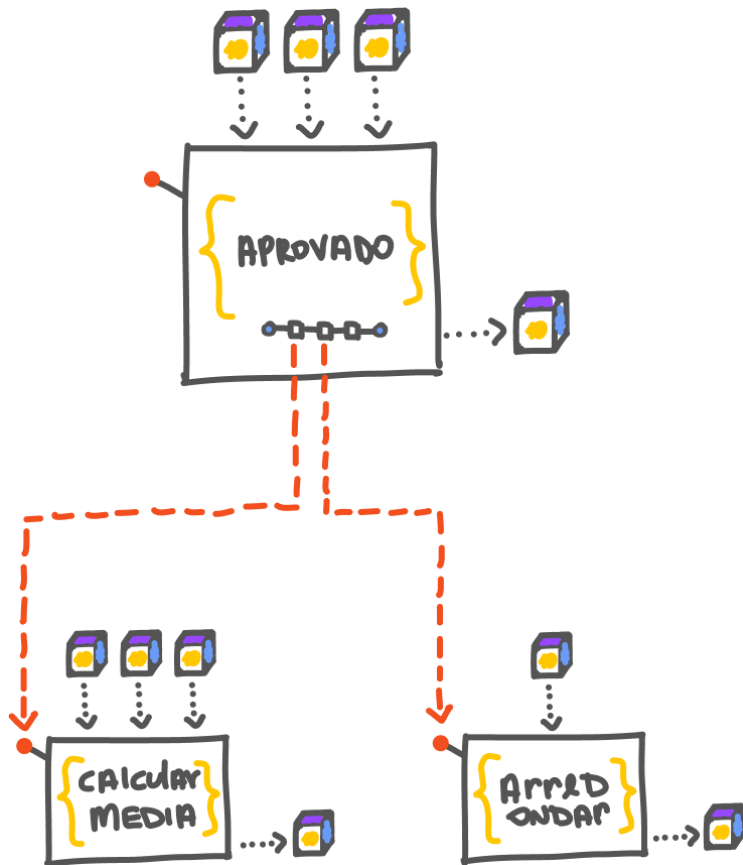
Chamamos de composição de funções, a habilidade de **perceber que tarefas grandes podem ser quebradas** em mini-tarefas servindo de auxílio para a tarefa principal e outras eventuais tarefas maiores, ao passo de também poderem ser executadas independentemente





Roteiro de Treino

em linguagens estáticas



```
>>
>>
>>
>> public double calcularMedia(double nota1, double nota2, double nota3)
>> {
>>     double media = (nota1 + nota2 + nota3) / 3;
>>     return media;
>> }
>>
>>
>>
>>
>> public long arredondar(double media)
>> {
>>     long mediaArredondada = Math.round(media);
>>     return mediaArredondada;
>> }
>>
>>
>>
>> public boolean Aprovado(double nota1, double nota2, double nota3)
>> {
>>     double media = calcularMedia(nota1, nota2, nota3);
>>     long mediaFinal = arredondar(media);
>>
>>     boolean passou = mediaFinal >= 5.0;
>>     return passou;
>> }
>>
>>
>>
```



ENCONTRO 06

Situações não previstas: Erros



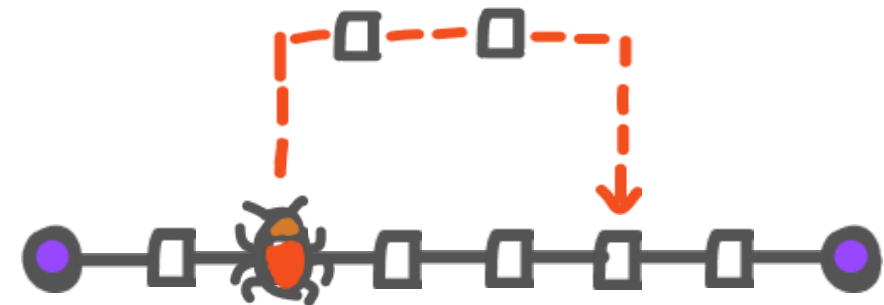
Agenda

- O que é uma Exception?
- Tipos de Erros
- Comando Try/Catch
- Exceptions Customizadas



Erros - Exceptions

Erros são **situações anormais** na execução do programa. Eles ocorrem quando uma situação não prevista acontece. É comum chamarmos os erros de um programa de **bugs**. Na vida de um programa, é inevitável que ocorram erros, por isso as linguagens de programação disponibilizam formas de **tratá-los**, ou seja, se um erro ocorrer, ao invés de encerrar a aplicação, o programa pode ser direcionado para um algoritmo que irá realizar ações para aquele tipo de erro.





Tipos de Exceptions

Existem três tipos de erros que devemos destacar: **compilação**, **execução** e **lógica**. Sendo que cada um reflete uma situação anormal para o programa.

- **Erro de Compilação:** Ocorre quando há um erro de sintaxe no código ou por *type check errors*. Esse tipo de erro impede que o programa seja executado.
- **Erro de Execução:** Ocorre quando uma função não consegue realizar sua ação geralmente devido aos valores de seus operandos. Também ocorrem quando a função percebe uma situação anormal para o algoritmo, que não se encaixa em um desvio, mas uma interrupção em sua continuação.
- **Erro de Lógica:** Ocorre quando o programa não se comporta como o esperado, ou seja, quando o resultado obtido não corresponde a lógica do problema. Esse tipo de erro não causa interrupção.





Roteiro de Treino

em linguagens estáticas – Java

Objetivos

- Conhecer os principais tipos de erros
- Entender os erros de Compilação
- Entender os erros de Execução
- Entender os erros de Lógica

```
>>
>>
>>
>>
>> // Erros de Compilação
>>
>> int x1 = 5 / "0";
>> int x2 = 5.trim();
>> int x3 = "0".trim();
>> int x4 = 5 / 5
>>
>>
>>
>> // Erros de Execução
>>
>> int x5 = 5 / 0;
>> String x6 = "Bruno".substring(5, 15);
>>
>>
>>
>> // Erros de Lógica
>>
>> double x7 = 4.0 + 6.0 + 8.0 / 3.0;
>> double x8 = (4.0 + 6.0 + 8.0) / 4.0;
>>
>>
>>
>>
```



Tratamento de Erro

Linguagem Java



Comando Try/Catch

em linguagens estáticas – Java

Garantem que se um erro de execução ocorrer, o programa não irá crashar/fechar. Nesse caso, ele será encaminhado para um algoritmo de desvio de erro.

Esse comando é composto de três partes:

- **O bloco try** envolve o algoritmo que será protegido.
- **O bloco catch** é chamada na ocasião de ocorrer um erro no algoritmo protegido no try.
- **O bloco finally** é opcional e sempre será executado, ocorrendo ou não um erro.

```
01
02
03
04
05
06
07 // Erro não tratado
08
09 public int integrantesPorGrupo(int pessoas, int grupos)
10 {
11     int qtd = pessoas / grupos;
12     return qtd;
13 }
```



```
14
15
16
17 int a = integrantesPorGrupo(30, 5);
18 System.out.println(a);
19
20
21
22 int b = integrantesPorGrupo(30, 0);
23 System.out.println(b);
24
25
26
27
28
29
```

```
PS C:\Users\Bruno\Code> java app.java
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at t.integrantesPorGrupo(t.java:15)
    at t.main(t.java:4)
```




Comando Try/Catch

em linguagens estáticas – Java

Garantem que se um erro de execução ocorrer, o programa não irá crashar/fechar. Nesse caso, ele será encaminhado para um algoritmo de desvio de erro.

Esse comando é composto de três partes:

- **O bloco try** envolve o algoritmo que será protegido.
- **O bloco catch** é chamada na ocasião de ocorrer um erro no algoritmo protegido no try.
- **O bloco finally** é opcional e sempre será executado, ocorrendo ou não um erro.

```
01
02
03
04
05
06 // Erros tratado:
07 // - Valor de retorno simbolizando erro
08
09 public int integrantesPorGrupo(int pessoas, int grupos)
10 {
11     try
12     {
13         int qtd = pessoas / grupos;
14         return qtd;
15     }
16     catch(Exception ex)
17     {
18         return -1;
19     }
20 }
```

```
21
22
23 int a = integrantesPorGrupo(30, 0);
24 System.out.println(a);
25
26
27
28
29
```

```
PS C:\Users\Bruno\Code> dotnet script app.csx
-1
```



Comando Try/Catch

em linguagens estáticas – Java

Garantem que se um erro de execução ocorrer, o programa não irá crashar/fechar. Nesse caso, ele será encaminhado para um algoritmo de desvio de erro.

Esse comando é composto de três partes:

- **O bloco try** envolve o algoritmo que será protegido.
- **O bloco catch** é chamada na ocasião de ocorrer um erro no algoritmo protegido no try.
- **O bloco finally** é opcional e sempre será executado, ocorrendo ou não um erro.

```
01
02
03 // Erros tratado:
04 // - Logando erro: Completo e por partes
05 // - Valor de retorno simbolizando erro
06
07 public int integrantesPorGrupo(int pessoas, int grupos)
08 {
09     try
10     {
11         int qtd = pessoas / grupos;
12         return qtd;
13     }
14     catch(Exception ex)
15     {
16
17         System.out.println(ex);
18         return -1;
19     }
20 }
21
22
23
24
25
26 int a = integrantesPorGrupo(30, 0);
27 System.out.println(a);
28
```

```
PS C:\Users\Bruno\Code> java app.java
> java.lang.ArithmeticException: / by zero
> -1
```



Comando Try/Catch

em linguagens estáticas – Java

Garantem que se um erro de execução ocorrer, o programa não irá crashar/fechar. Nesse caso, ele será encaminhado para um algoritmo de desvio de erro.

Esse comando é composto de três partes:

- **O bloco try** envolve o algoritmo que será protegido.
- **O bloco catch** é chamada na ocasião de ocorrer um erro no algoritmo protegido no try.
- **O bloco finally** é opcional e sempre será executado, ocorrendo ou não um erro.

```
01
02
03
04
05 // Erros tratado:
06 // - Logando erro
07 // - Relançando erro tratado
08
09 public int integrantesPorGrupo(int pessoas, int grupos)
10 {
11     try
12     {
13         int qtd = pessoas / grupos;
14         return qtd;
15     }
16     catch(Exception ex)
17     {
18         System.out.println(ex.getMessage());
19         throw ex;
20     }
21 }
22
23
24 int a = integrantesPorGrupo(30, 0);
25 System.out.println(a);
26
27
28
```

```
PS C:\Users\Bruno\Code> java app.java
/ by zero
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at t.integrantesPorGrupo(t.java:11)
    at t.main(t.java:4)
```



Comando Try/Catch

em linguagens estáticas – Java

Garantem que se um erro de execução ocorrer, o programa não irá crashar/fechar. Nesse caso, ele será encaminhado para um algoritmo de desvio de erro.

Esse comando é composto de três partes:

- **O bloco try** envolve o algoritmo que será protegido.
- **O bloco catch** é chamada na ocasião de ocorrer um erro no algoritmo protegido no try.
- **O bloco finally** é opcional e sempre será executado, ocorrendo ou não um erro.

```
01
02
03 // Erros tratado:
04 // - Catch aninhado
05 // - O catch é executado dependendo do erro
06 // - O catch com tipo Exception é o mais geral
07 // - Apenas um catch pode ser executado
08
09 public int integrantesPorGrupo(int pessoas, int grupos)
10 {
11     try
12     {
13         int qtd = pessoas / grupos;
14         return qtd;
15     }
16     catch (Exception ex)
17     {
18         return 0;
19     }
20
21
22
23
24 }
25
26
27 int a = integrantesPorGrupo(30, 0);
28 System.out.println(a);
29
```

```
PS C:\Users\Bruno\Code> java app.java
0
```



Comando Try/Catch

em linguagens estáticas – Java

Garantem que se um erro de execução ocorrer, o programa não irá crashar/fechar. Nesse caso, ele será encaminhado para um algoritmo de desvio de erro.

Esse comando é composto de três partes:

- **O bloco try** envolve o algoritmo que será protegido.
- **O bloco catch** é chamada na ocasião de ocorrer um erro no algoritmo protegido no try.
- **O bloco finally** é opcional e sempre será executado, ocorrendo ou não um erro.

```
01
02 // Erros tratado:
03 // - Forçando um novo Erro
04 public int integrantesPorGrupo(int pessoas, int grupos)
05 {
06     try
07     {
08         if (pessoas <= 0)
09             throw new IllegalArgumentException(
10                 "Qtd de pessoas inválida!");
11
12         int qtd = pessoas / grupos;
13         return qtd;
14     }
15     catch (IllegalArgumentException ex)
16     {
17         System.out.println(ex.getMessage());
18     }
19     return 0;
20 }
21
22
23
24
25 }
26
27 int a = integrantesPorGrupo(0, 6);
28 System.out.println(a);
29
```

```
PS C:\Users\Bruno\Code> dotnet script app.csx
Qtd de pessoas inválida!
0
```



Comando Try/Catch

em linguagens estáticas – Java

Garantem que se um erro de execução ocorrer, o programa não irá crashar/fechar. Nesse caso, ele será encaminhado para um algoritmo de desvio de erro.

Esse comando é composto de três partes:

- **O bloco try** envolve o algoritmo que será protegido.
- **O bloco catch** é chamada na ocasião de ocorrer um erro no algoritmo protegido no try.
- **O bloco finally** é opcional e sempre será executado, ocorrendo ou não um erro.

```
01 // Erros tratado:
02 // - Forçando erros para situações específicas.
03 public int integrantesPorGrupo(int pessoas, int grupos)
04 {
05     try
06     {
07         if (pessoas <= 0)
08             throw new IllegalArgumentException(
09                 "Qtd de pessoas inválida!");
10
11         if (grupos <= 0)
12             throw new IllegalArgumentException(
13                 "Qtd de grupos inválida!");
14
15         int qtd = pessoas / grupos;
16         return qtd;
17     }
18     catch (ArgumentException ex)
19     {
20         System.out.println(ex.getMessage());
21         return 0;
22     }
23     catch (Exception ex)
24     {
25         return -1;
26     }
27 }
28
29
```

```
PS C:\Users\Bruno\Code> java app.java
Qtd de grupos inválida!
0
```



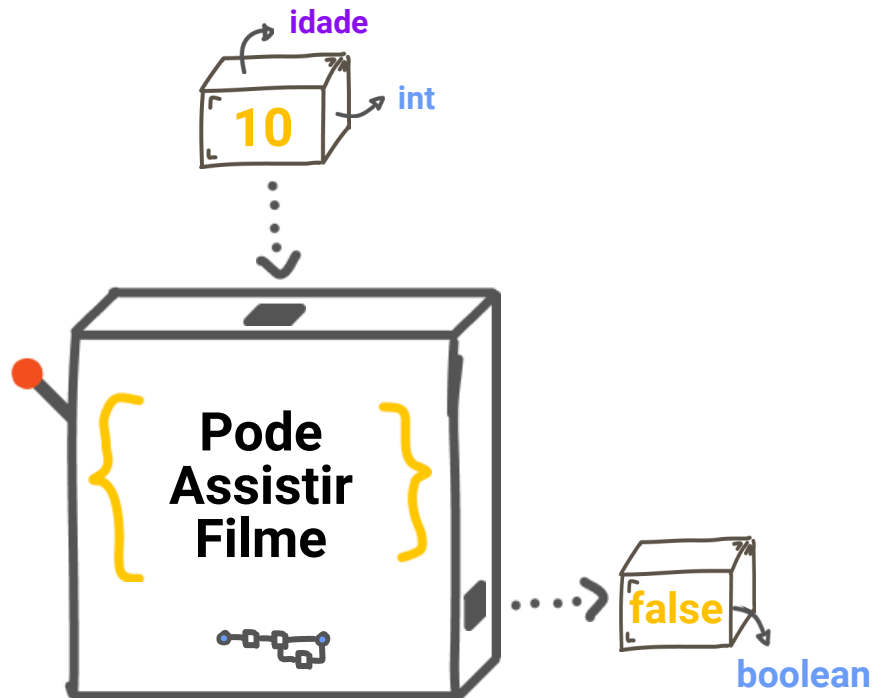
Tratamento de Erro

Validando Entrada de Dados



Máquina de Função (PodeAssistirFilme)

em linguagens estáticas



```
01
02
03
04
05
06
07
08 public boolean podeAssistirFilme(int idade)
09 {
10     try
11     {
12         if (idade <= 0)
13             throw new IllegalArgumentException(
14                 "Você não tinha nascido");
15
16         if (idade >= 110)
17             throw new IllegalArgumentException(
18                 "Dúvido chegar nessa idade");
19
20         boolean pode = idade >= 12;
21         return pode;
22     }
23     catch (Exception ex)
24     {
25         System.out.println(ex.getMessage());
26         return false;
27     }
28 }
29
30
31
32 boolean x = podeAssistirFilme(10);
33 System.out.println(x);
34
35
36
```




ENCONTRO 08

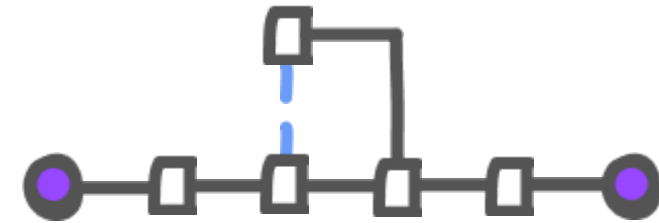
Caminhos Algorítmicos - Decisões



Desvios condicionais

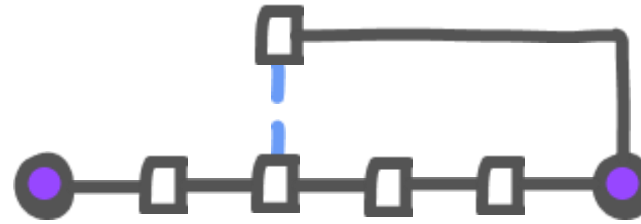
Um algoritmo nem sempre é um caminho reto. Ele pode tomar **caminhos diferentes** dependendo da entrada que lhe foi dada, enquanto entradas iguais geralmente percorrem o mesmo caminho.

Na programação, podemos determinar os variados caminhos que um algoritmo pode tomar através de **comandos condicionais**. Esses comandos permitem que o programador planeje seu algoritmo de forma que resolva o problema em **contextos diferentes**.

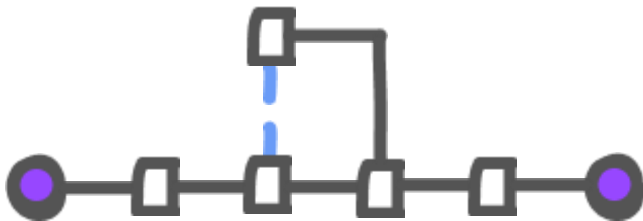




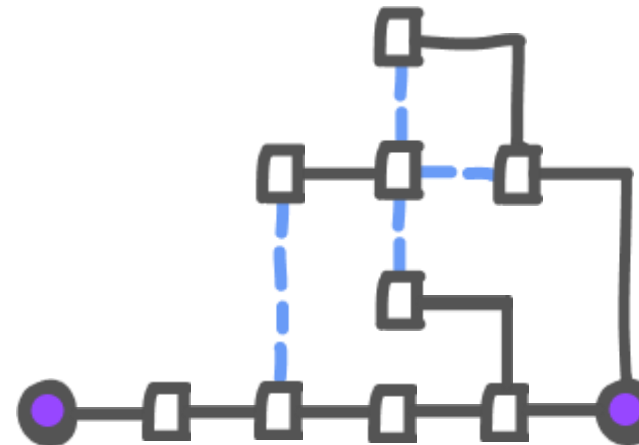
Caminho Decisão Simples



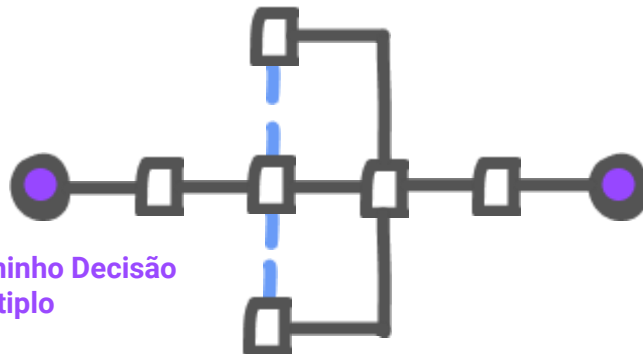
Caminho Decisão com Interrupção



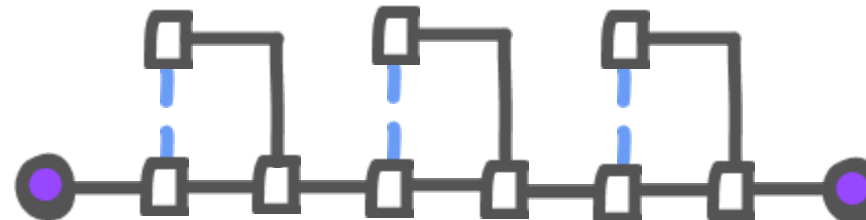
Caminho Decisão Opcional



Caminho Decisão Aninhada



Caminho Decisão Múltiplo



Caminho Decisão Sequencial

Tipos de Caminhos

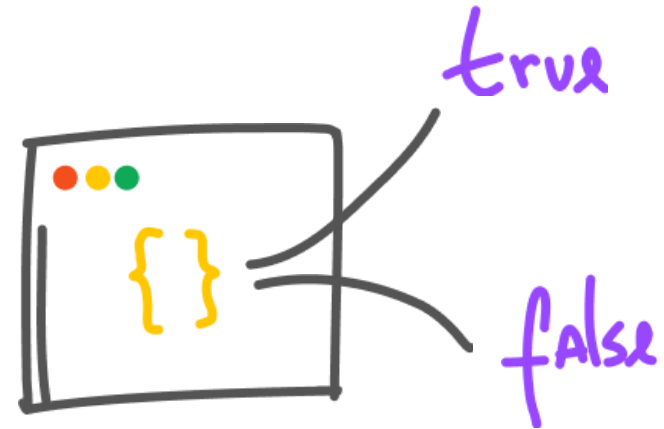


Condição

Decisões acontecem a partir de uma **condição**, ou seja, **uma expressão que retorne verdadeiro ou falso**. Essa expressão é colocada no comando condicional que decide qual caminho seguir, em outras palavras, qual bloco será executado.

Condições podem ser:

- Variáveis booleanas;
- Resultado de operadores relacionais ou lógicos;
- Chamadas de funções que retornam booleanos.



```
>> ...
>>
>> boolean p1 = true;
>> boolean p2 = n > 4;
>> boolean p3 = n > 4 || p == "dev";
>> boolean p4 = p.contains("e");
>> boolean p5 = par(4);
>>
>> public boolean par(int numero)
>> {
>>     return numero % 2 == 0;
>> }
```



Limitações Caminho Único

Algoritmos de Caminho Único possuem **grandes limitações** para resolução de **problemas complexos**, ou seja, que possuem **regras específicas** que devem ser seguidas dependendo do contexto atual que está sendo executado.

Um algoritmo de caminho único não pode possuir variação em suas regras, ele **sempre seguirá as mesmas instruções e passará sempre pelas mesmas computações**, ele pode ser pensado como um processo simples.

Um algoritmo com decisões é utilizado para resolver problemas que possuem **regras variadas** que são executadas **dependendo da situação do programa**.

01
02
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35

```
public boolean passou(double nota1, double nota2, double nota3)
{
    double media = (nota1 + nota2 + nota3) / 3;
    boolean p = media >= 5;
    return p;
}
```



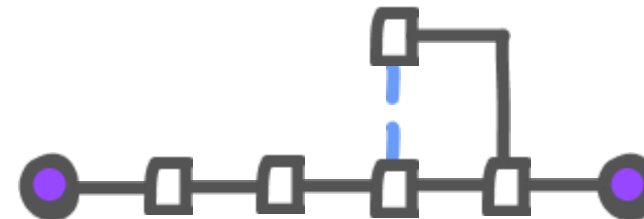
Comando If/Else If/Else

O comando IF é o comando de condição que possibilita a variação de regras em um algoritmo, ele é o mais utilizado e que está disponível em todas linguagens atuais. O comando IF oferece **três variações** de seu uso.

PRIMEIRA VARIAÇÃO

É composta do bloco IF que será executado se a expressão booleana que fica em seus parênteses (Condição) for **verdadeira**.

```
01  
02  
03 public String passou(double nota1, double nota2, double nota3)  
04 {  
05     double media = (nota1 + nota2 + nota3) / 3;  
06  
07     String situacao = "Reprovado";  
08     if (media >= 5)  
09     {  
10         situacao = "Aprovado";  
11     }  
12     return situacao;  
13 }  
14
```





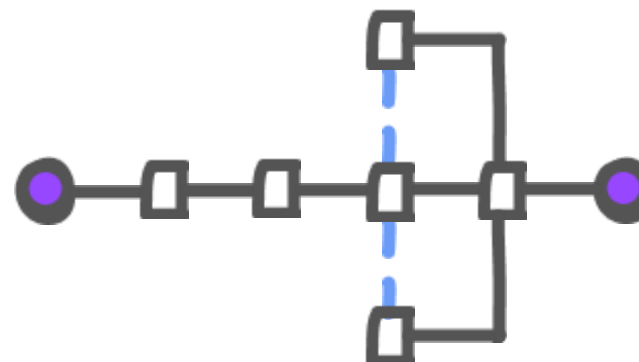
Comando If/Else If/Else

O comando IF é o comando de condição que possibilita a variação de regras em um algoritmo, ele é o mais utilizado e que está disponível em todas linguagens atuais. O comando IF oferece **três variações** de seu uso.

SEGUNDA VARIAÇÃO

É composta de um **bloco IF** que será executado se a condição for **verdadeira**, e de outro **bloco ELSE**, que é executado se a condição for **falsa**. Os dois blocos nunca serão executados na mesma chamada, será decidido entre ou outro.

```
01  
02 public String passou(double nota1, double nota2, double nota3)  
03 {  
04     double media = (nota1 + nota2 + nota3) / 3;  
05  
06     String situacao = "";  
07     if (media >= 5)  
08     {  
09         situacao = "Aprovado";  
10     }  
11     else  
12     {  
13         situacao = "Reprovado";  
14     }  
15     return situacao;  
16 }  
17
```





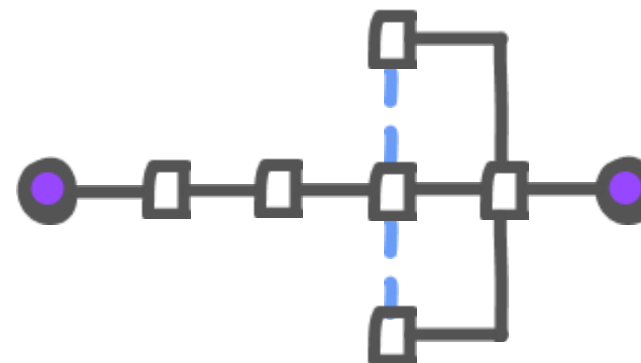
Comando If/Else If/Else

O comando IF é o comando de condição que possibilita a variação de regras em um algoritmo, ele é o mais utilizado e que está disponível em todas linguagens atuais. O comando IF oferece **três variações** de seu uso.

TERCEIRA VARIAÇÃO

É composta composta de várias **condições intermediárias** chamados **ELSE IF** que são testados sequencialmente **caso seu antecessor seja falso**. Caso uma condição seja verdadeira, seu bloco é executado e os demais são ignorados. Caso todos predicados sejam falsos, é executado o bloco else, caso houver.

```
01  
02 public String passou(double nota1, double nota2, double nota3)  
03 {  
04     double media = (nota1 + nota2 + nota3) / 3;  
05  
06     String situacao = "";  
07     if (media >= 5)  
08     {  
09         situacao = "Aprovado";  
10     }  
11     else if (media >= 3)  
12     {  
13         situacao = "Recuperação";  
14     }  
15     else  
16     {  
17         situacao = "Reprovado";  
18     }  
19     return situacao;  
20 }  
21
```





Comando Switch/Case

O comando Switch também disponível em diversas linguagens, com raras exceções como Python, possui **dois elementos principais**.

O **primeiro** é designado pelo próprio nome de **switch** [06-35] e objetiva selecionar uma variável para servir de referência para comparação de igualdade. A **segunda** parte são os chamados **cases** [8,12,16,20,24,28,32] que são os valores à serem comparados com o valor da variável selecionada no switch.

Os cases são testados **sequencialmente** e quando o valor de um case for igual ao valor selecionado na variável do switch, **seu bloco é executado e todos cases seguintes são ignorados**. Se nenhum case for executado, o bloco **default** [36] será executado.

```
01
02 public String diaSemana(int dia)
03 {
04     String nome = "";
05
06     switch (dia)
07     {
08         case 1:
09             nome = "Domingo";
10             break;
11
12         case 2:
13             nome = "Segunda-Feira";
14             break;
15
16         case 3:
17             nome = "Terça-Feira";
18             break;
19
20         case 4:
21             nome = "Quarta-Feira";
22             break;
23
24         case 5:
25             nome = "Quinta-Feira";
26             break;
27
28         case 6:
29             nome = "Sexta-Feira";
30             break;
31
32         case 7:
33             nome = "Sábado";
34             break;
35
36         default:
37             nome = "Inválido";
38             break;
39     }
40
41     return nome;
42 }
```



CONDIÇÃO

Tipos de Caminho



Caminho Simples

O algoritmo que possui caminho simples é aquele que sempre executará todas as instruções da função. Ele é simples justamente por não ter desvios, **seu caminho é único**.

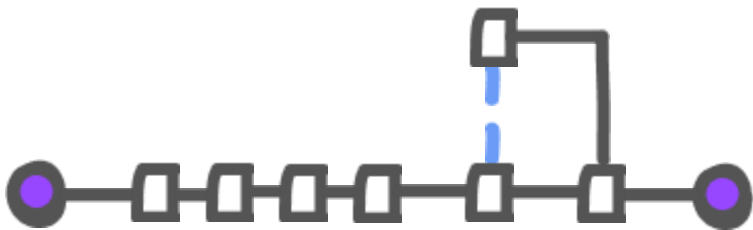


```
01  
02  
03  
04  
05  
06  
07  
08  
09  
10  
11  
12 public boolean areaRetangulosIguais(int base1, int altura1,  
13                                     int base2, int altura2,  
14                                     int base3, int altura3)  
15 {  
16     int area1 = base1 * altura1;  
17     int area2 = base2 * altura2;  
18     int area3 = base3 * altura3;  
19  
20     boolean iguais = area1 == area2 && area2 == area3;  
21     return iguais;  
22 }  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35
```



Caminho Decisão Opcional

O algoritmo que possui caminho decisão opcional é aquele que durante a execução das instruções da função, **decide se um bloco de código será executado ou não [21:24]**, e logo depois, segue seu caminho. Nesse tipo de caminho pode acontecer do bloco não ser executado.

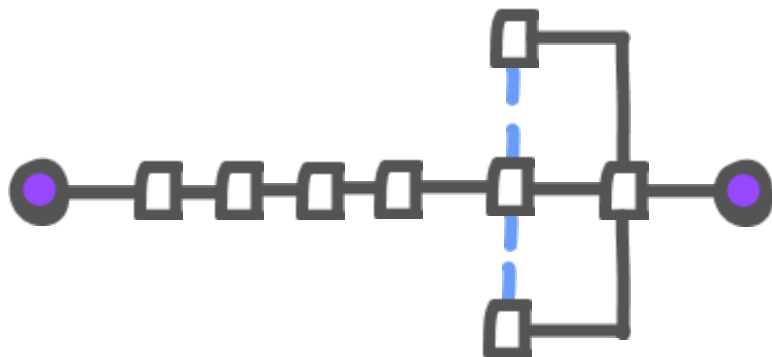


```
01  
02  
03  
04  
05  
06  
07  
08  
09  
10  
11  
12 public String areaRetangulosIguais(int base1, int altura1,  
13                                     int base2, int altura2,  
14                                     int base3, int altura3)  
15 {  
16     int area1 = base1 * altura1;  
17     int area2 = base2 * altura2;  
18     int area3 = base3 * altura3;  
19  
20     String msgm = "Retângulos diferentes";  
21     if (area1 == area2 && area2 == area3)  
22     {  
23         msgm = "Retângulos iguais";  
24     }  
25  
26     return msgm;  
27 }  
28  
29  
30  
31  
32  
33  
34  
35
```



Caminho Decisão Múltipla

O algoritmo que possui caminho decisão múltiplo é aquele que durante a execução das instruções da função, **decide executar UM entre diversos blocos**. Nesse tipo de caminho, sempre UM e no máximo UM dos blocos será executado.

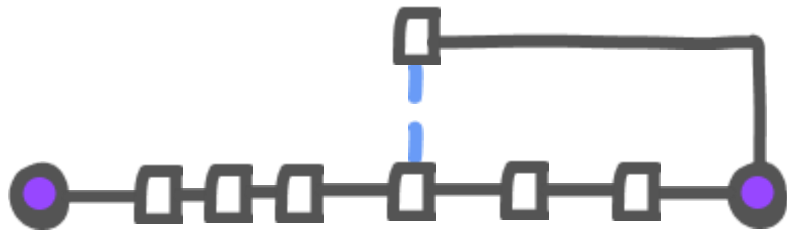


```
01
02
03
04
05
06
07 public String areaRetangulosIguais(int base1, int altura1,
08                                     int base2, int altura2,
09                                     int base3, int altura3)
10 {
11     int area1 = base1 * altura1;
12     int area2 = base2 * altura2;
13     int area3 = base3 * altura3;
14
15     String msgm = "";
16     if (area1 == area2 && area2 == area3)
17     {
18         msgm = "Todos retângulos são iguais";
19     }
20     else if (area1 == area2 || area1 == area3 || area2 == area3)
21     {
22         msgm = "Dois retângulos são iguais";
23     }
24     else
25     {
26         msgm = "Nenhum retângulo é igual";
27     }
28
29     return msgm;
30 }
31
32
33
34
35
```



Caminho Decisão Interrupção

O algoritmo que possui caminho decisão com interrupção é aquele que durante a execução das instruções da função, **decide se um bloco de código será executado ou não**. A diferença para o caminho opcional é que o bloco de código da decisão, **antecipa a resposta da função, fazendo com que os códigos seguintes não sejam executados**.

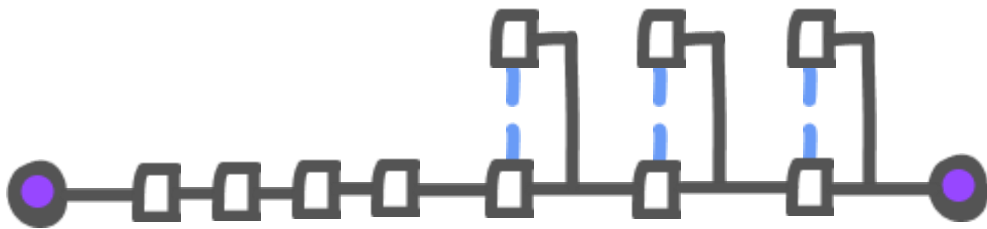


```
01  
02  
03  
04  
05  
06  
07  
08  
09  
10  
11 public boolean areaRetangulosIguais(int base1, int altura1,  
12                                     int base2, int altura2,  
13                                     int base3, int altura3)  
14 {  
15     int area1 = base1 * altura1;  
16     int area2 = base2 * altura2;  
17     int area3 = base3 * altura3;  
18  
19     if (area1 <= 0 || area2 <= 0 || area3 <= 0)  
20     {  
21         return false;  
22     }  
23  
24     boolean iguais = area1 == area2 && area2 == area3;  
25     return iguais;  
26 }  
27  
28  
29  
30  
31  
32  
33  
34  
35
```



Caminho Decisão em Sequência

O algoritmo que possui caminho com decisão em sequência é aquele que durante a execução das instruções da função, possui uma **sequência de decisões independentes, ou seja, vários blocos de código podem ser executados em seguida, já que cada decisão não possui relação com a próxima.**

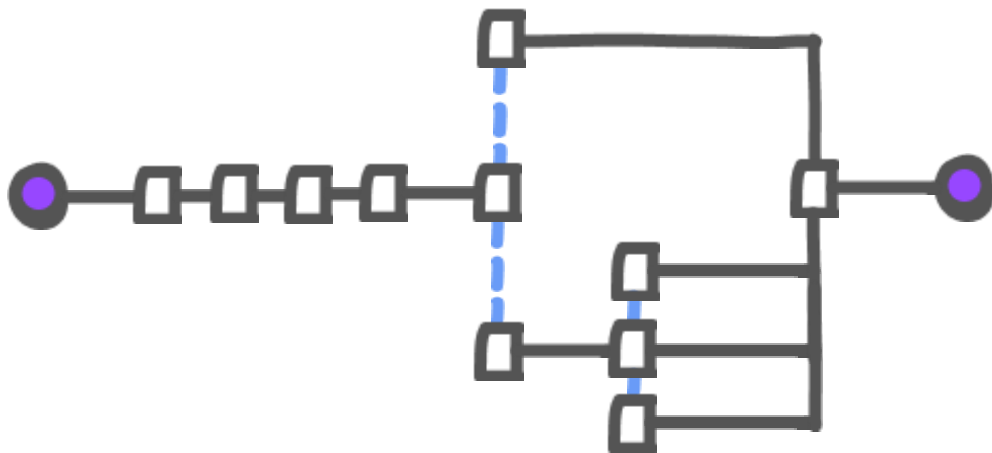


```
01
02
03
04
05
06 public String areaRetangulosIguais(int base1, int altura1,
07                                     int base2, int altura2,
08                                     int base3, int altura3)
09 {
10     int area1 = base1 * altura1;
11     int area2 = base2 * altura2;
12     int area3 = base3 * altura3;
13
14     String msgm = "";
15     if (area1 == area2)
16     {
17         msgm = msgm + "[R1=R2] ";
18     }
19
20     if (area1 == area3)
21     {
22         msgm = msgm + "[R1=R3] ";
23     }
24
25     if (area2 == area3)
26     {
27         msgm = msgm + "[R2=R3] ";
28     }
29
30     return msgm;
31 }
32
33
34
35
```



Caminho Decisão Aninhado

O algoritmo que possui caminho com decisão aninhado é aquele que durante a execução das instruções da função, **possui vários desvios UM dentro do OUTRO**, aumentando sua complexidade já que as possibilidades dos caminhos aumentam linearmente.



```
01
02
03
04
05
06 public String areaRetangulosIguais(int base1, int altura1,
07                                     int base2, int altura2,
08                                     int base3, int altura3)
09 {
10     int area1 = base1 * altura1;
11     int area2 = base2 * altura2;
12     int area3 = base3 * altura3;
13
14     String msgm = "";
15     if (area1 == area2 && area2 == area3)
16     {
17         msgm = "Todos retângulos iguais";
18     }
19     else
20     {
21         if (area1 == area2)
22             msgm = "Retângulo 1 igual ao 2";
23         else if (area1 == area3)
24             msgm = "Retângulo 1 igual ao 3";
25         else if (area2 == area3)
26             msgm = "Retângulo 2 igual ao 3";
27         else
28             msgm = "Nenhum retângulo é igual";
29     }
30
31     return msgm;
32 }
33
34
35
```

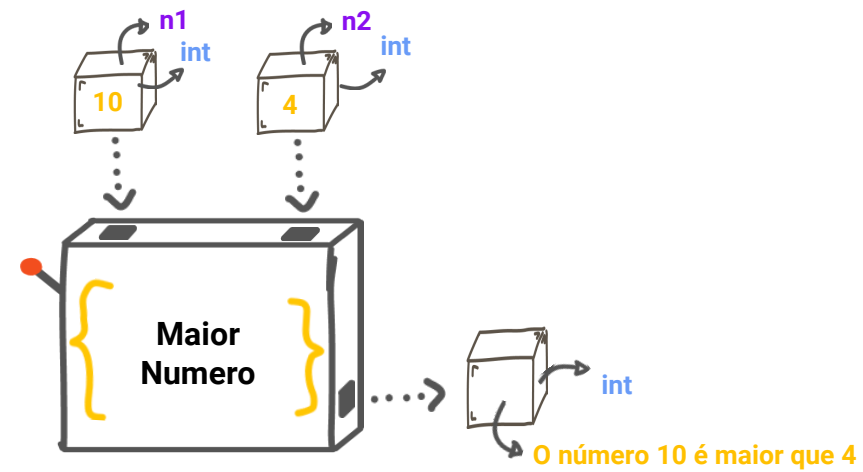



CONDIÇÃO

Treino Prático



Máquina de Função (Maior Número) em linguagens estáticas



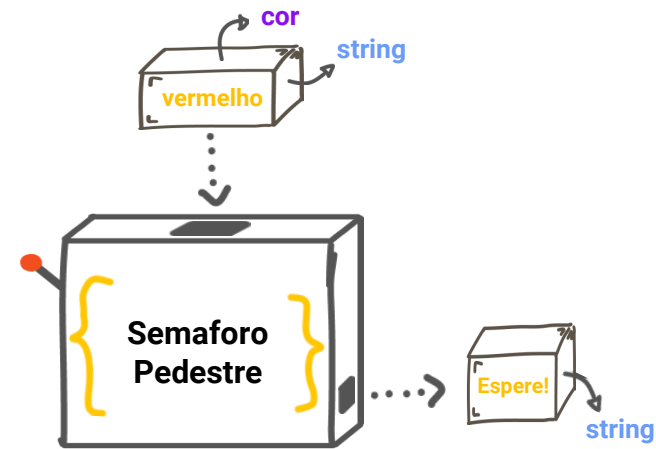
```
01 public String maiorNumero(int n1, int n2)
02 {
03     String msgm = "";
04
05     if (n1 > n2)
06     {
07         msgm = "O número " + n1 + " é maior que " + n2;
08     }
09     else if (n1 < n2)
10     {
11         msgm = "O número " + n2 + " é maior que " + n1;
12     }
13     else
14     {
15         msgm = "Os números são iguais";
16     }
17
18     return msgm;
19 }
20
21
22
23
24
```

```
25 import java.util.Scanner;
26
27 public void Main(String[] args)
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa MAIOR NÚMERO ##");
32
33         System.out.println("Informe um número:");
34         int a = input.nextInt();
35
36         System.out.println("Informe outro número:");
37         int b = input.nextInt();
38
39         String x = maiorNumero(a, b);
40
41         System.out.println(x);
42     }
43     catch(Exception ex) {
44         System.out.println("Ops, ocorreu um erro:");
45         System.out.println(ex.getMessage());
46     }
47 }
48
```



Máquina de Função (Semaforo de Pedestre)

em linguagens estáticas



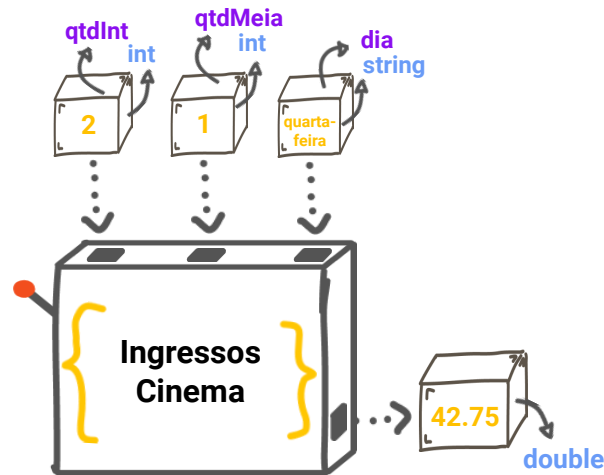
```
01
02
03 public String semaforoPedestre(String cor)
04 {
05     if (cor != "verde" && cor != "vermelho")
06         throw new IllegalArgumentException("Semáforo inoperante");
07
08     String msgm = "Espere!";
09
10     if (cor == "verde")
11     {
12         msgm = "Atravesse!";
13     }
14
15     return msgm;
16 }
17
18
19
20
21
22
23
24
```

```
25 import java.util.Scanner;
26
27 public void Main()
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa SEMÁFORO PEDESTRE ##");
32
33         System.out.println("Informe a cor do semáforo:");
34         String a = input.nextLine();
35
36         String x = semaforoPedestre(a);
37
38         System.out.println(x);
39     }
40     catch(Exception ex) {
41         System.out.println("Ops, ocorreu um erro:");
42         System.out.println(ex.getMessage());
43     }
44 }
45
46
47
48
```



Máquina de Função (Ingresso Cinema)

em linguagens estáticas



```
01 public double ingressosCinema(int qtdInteira, int qtdMeia, String dia)
02 {
03     double total = 0;
04
05     if (dia == "quarta-feira")
06     {
07         total = (qtdInteira + qtdMeia) * 14.25;
08     }
09     else
10     {
11         total = (qtdMeia * 14.25) + (qtdInteira * 28.5);
12     }
13
14     return total;
15 }
16
17
18
```

```
19 import java.util.Scanner;
20
21
22 public void Main()
23 {
24     try {
25         Scanner input = new Scanner(System.in);
26         System.out.println("## Programa INGRESSO ##");
27
28         System.out.println("Informe a qtd. de Inteiras:");
29         int a = input.nextInt();
30
31         System.out.println("Informe a qtd. de Meias:");
32         int b = input.nextInt();
33
34         System.out.println("Informe o dia da semana do filme:");
35         String c = input.nextLine();
36
37         double x = ingressosCinema(a, b, c);
38
39         System.out.println("O total da compra é de R$ " + x);
40     }
41     catch(Exception ex) {
42         System.out.println("Ops, ocorreu um erro:");
43         System.out.println(ex.getMessage());
44     }
45 }
46
47
48
49
```



ENCONTRO 09

Caminhos Algorítmicos - Repetições

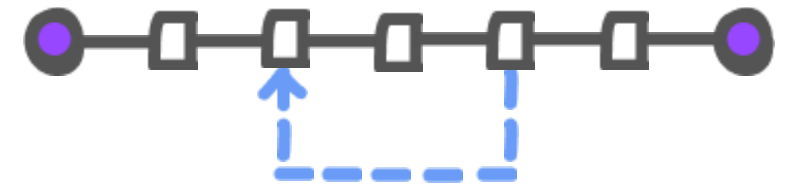


Repetições

Como vimos, um algoritmo nem sempre é um caminho reto, ele pode tomar **caminhos diferentes** dependendo da entrada que lhe foi dada, vimos que com **caminhos condicionais** podemos resolver problemas mais complexos já que diferentes regras podem ser consideradas no mesmo algoritmo.

Sendo a linguagem de programação executada em uma máquina, sua vantagem está em realizar **milhões de cálculos por segundo**. *Se pensarmos que cada linha de código realiza um cálculo, teríamos então que escrever 1 milhão de linhas de código para o computador realizar 1 milhão de cálculos?*

Isso não parece fazer sentido e de fato não faz, **problemas grandes geralmente realizam os mesmos cálculos milhares de vezes**, mas não precisamos reescrever essas instruções nesse quantidade, isso porque toda linguagem de programação disponibiliza **comandos de repetição** que permitem programar uma sequência de cálculos para ser executada milhões de vezes.

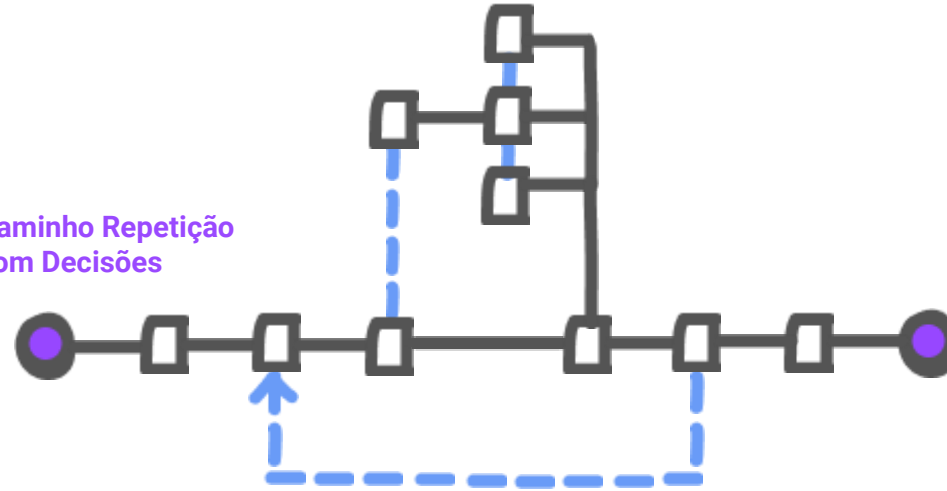




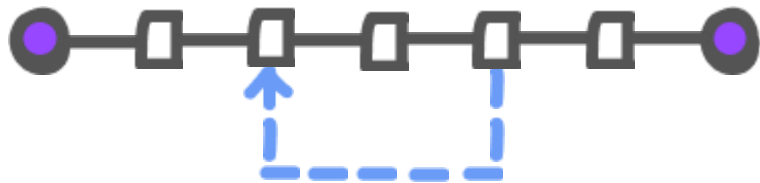
Caminho Simples



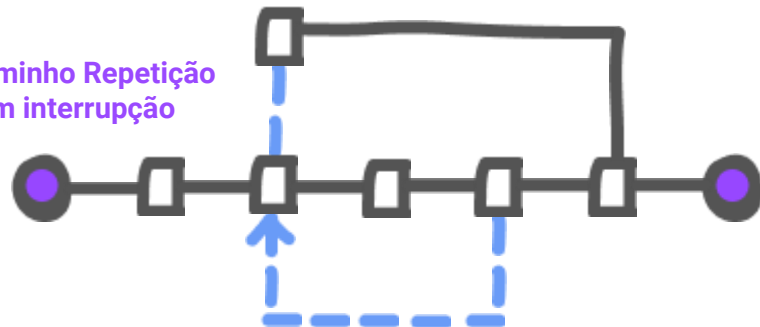
Caminho Repetição com Decisões



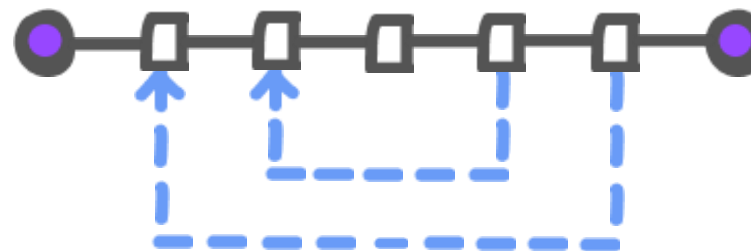
Caminho Repetição Simples



Caminho Repetição com interrupção



Caminho Repetição aninhada



Tipos de Caminhos



Limitações

Algoritmos de caminho reto ou com decisões (condicionais) resolvem uma grande variedade de problemas computacionais, mas ainda estão **limitados em executar no máximo, a quantidade de operações baseada na quantidade de linhas de código** que o programador escreve.

Algoritmos de **complexidade superior utilizam-se de caminhos com repetição**, os quais através de poucas linhas de código, podem executar milhões de operações.

Com essa nova classe de caminhos algoritmos, o programador completa em sua mochila de ferramentas, os recursos necessários para **resolver os problemas do mais alto grau de complexidade**.

```
01
02
03
04 // Insanidade
05 // Contar de 1 a 1 milhão com 1 milhão de linhas
06
07 int contador = 1;
08 System.out.println(contador);
09
10 contador = 2;
11 System.out.println(contador);
12
13 contador = 3;
14 System.out.println(contador);
15
16 contador = 4;
17 System.out.println(contador);
18
19 ...
20
21
22 // Forma passível de ser repetida
23
24 int contador = 1;
25 System.out.println(contador);
26
27 contador = contador + 1;
28 System.out.println(contador);
29
30 contador = contador + 1;
31 System.out.println(contador);
32
33 contador = contador + 1;
34 System.out.println(contador);
35
```




Limitações

Algoritmos de caminho reto ou com decisões (condicionais) resolvem uma grande variedade de problemas computacionais, mas ainda estão **limitados em executar no máximo, a quantidade de operações baseada na quantidade de linhas de código** que o programador escreve.

Algoritmos de **complexidade superior utilizam-se de caminhos com repetição**, os quais através de poucas linhas de código, podem executar milhões de operações.

Com essa nova classe de caminhos algoritmos, o programador completa em sua mochila de ferramentas, os recursos necessários para **resolver os problemas do mais alto grau de complexidade**.

```
01
02
03
04 // Forma resumida de incrementação
05
06 int contador = 1;
07 System.out.println(contador);
08
09 contador++;
10 System.out.println(contador);
11
12 contador++;
13 System.out.println(contador);
14
15 contador++;
16 System.out.println(contador);
17
18 ...
19
20
21
22 // Pseudo-código: Comando de repetição limitado
23
24 int contador = 1;
25 enquanto (contador <= 10) // pseudo-comando
26 {
27     System.out.println(contador);
28     contador++;
29 }
30
31
32
33
34
35
```



Limitações

Algoritmos de caminho reto ou com decisões (condicionais) resolvem uma grande variedade de problemas computacionais, mas ainda estão **limitados em executar no máximo, a quantidade de operações baseada na quantidade de linhas de código** que o programador escreve.

Algoritmos de **complexidade superior utilizam-se de caminhos com repetição**, os quais através de poucas linhas de código, podem executar milhões de operações.

Com essa nova classe de caminhos algoritmos, o programador completa em sua mochila de ferramentas, os recursos necessários para **resolver os problemas do mais alto grau de complexidade**.

```
01
02
03
04
05
06
07
08
09
10
11 // Jeito correto
12 // Contar de 1 a 1 milhão utilizando comando de repetição
13
14
15 int contador = 1;
16 while ( contador <= 1000000 )
17 {
18     System.out.println(contador);
19     contador++;
20 }
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```



Limitações

Algoritmos de caminho reto ou com decisões (condicionais) resolvem uma grande variedade de problemas computacionais, mas ainda estão **limitados em executar no máximo, a quantidade de operações baseada na quantidade de linhas de código** que o programador escreve.

Algoritmos de **complexidade superior utilizam-se de caminhos com repetição**, os quais através de poucas linhas de código, podem executar milhões de operações.

Com essa nova classe de caminhos algoritmos, o programador completa em sua mochila de ferramentas, os recursos necessários para **resolver os problemas do mais alto grau de complexidade**.

```
01
02
03
04 // Insanidade
05 // Contar de 1 milhão a 1 com 1 milhão de linhas
06
07 int contador = 1000000;
08 System.out.println(contador);
09
10 contador--;
11 System.out.println(contador);
12
13 contador--;
14 System.out.println(contador);
15
16 contador--;
17 System.out.println(contador);
18
19 ...
20
21
22 // Jeito correto
23 // Contar de 1 milhão a 1 utilizando comando de repetição
24
25 int contador = 1000000;
26 while (contador > 0)
27 {
28     System.out.println(contador);
29     contador--;
30 }
31
32
33
34
35
```



Roteiro de Treino

Objetivos

- Aprimorar a leitura de código;
- Compreender o processo de repetição;
- Prever o comportamento da máquina;
- Compreender comando de incrementação;
- Compreender comando de decrementação;
- Contagem crescente;
- Contagem decrescente;

```
01
02 // Treino 01
03 int n = 5;
04 while (n <= 10)
05 {
06     System.out.println(n);
07     n++;
08 }
09
10
11 // Treino 02
12 int n = 1;
13 while (n <= 10)
14 {
15     System.out.println(n);
16     n+=2;
17 }
18
19
20 // Treino 03
21 int n = 9;
22 while (n < 10)
23 {
24     System.out.println(n);
25     n++;
26 }
27
28
29 // Treino 04
30 int n = 1;
31 while (n <= 2)
32 {
33     System.out.println(n);
34     n++;
35 }
```



Roteiro de Treino

Objetivos

- Aprimorar a leitura de código;
- Compreender o processo de repetição;
- Prever o comportamento da máquina;
- Compreender comando de incrementação;
- Compreender comando de decrementação;
- Contagem crescente;
- Contagem decrescente;

```
01
02 // Treino 05
03 int n = 5;
04 while (n >= 1)
05 {
06     System.out.println(n);
07     n--;
08 }
09
10
11 // Treino 06
12 int n = 10;
13 while (n > 0)
14 {
15     System.out.println(n);
16     n-=5;
17 }
18
19
20 // Treino 07
21 int n = 10;
22 while (n > 8)
23 {
24     System.out.println(n);
25     n--;
26 }
27
28
29 // Treino 08
30 int n = 1;
31 while (n > 0)
32 {
33     System.out.println(n);
34     n++;
35 }
```



Roteiro de Treino

Objetivos

- Relembrar o uso de funções;
- Utilizar repetições em função;
- Introduzir a ideia de funções sem retorno
- Exemplos

```
01
02
03
04
05
06 // Treino 09
07
08 public void contarAte(int limite)
09 {
10     int cont = 1;
11     while (cont <= limite)
12     {
13         System.out.println(cont);
14         cont++;
15     }
16 }
17
18
19
20
21
22 // Treino 10
23
24 public void contarAte(int limite)
25 {
26     int cont = limite;
27     while (cont >= 1)
28     {
29         System.out.println(cont);
30         cont--;
31     }
32 }
33
34
35
```



Roteiro de Treino

Objetivos

- Ideia de contagem;
- Ideia de repetição;
- Desenhar formas para representação visual
- Compreender o processo de repetir
- Compreender o processo de repetir processos de repetição

```
01
02
03
04 // Repetição de palavras
05 int n = 1;
06 while (n <= 5)
07 {
08     System.out.println("Bruno");
09     n++;
10 }
11
12
13
14
15 // Linha Vertical
16 int n = 1;
17 while (n <= 5)
18 {
19     System.out.println("* ");
20     n++;
21 }
22
23
24
25
26 // Linha Horizontal
27 int n = 1;
28 while (n <= 5)
29 {
30     System.out.print("* ");
31     n++;
32 }
33
34
35
```



Roteiro de Treino

Objetivos

- Ideia de contagem;
- Ideia de repetição;
- Desenhar formas para representação visual
- Compreender o processo de repetir
- Compreender o processo de repetir processos de repetição

// Quadrado

```
int linha = 1;
while (linha <= 5)
{
    int coluna = 1;
    while (coluna <= 5)
    {
        System.out.print("* ");
        coluna++;
    }

    System.out.println();
    linha++;
}
```




REPETIÇÃO

Roteiro - parte II



Roteiro de Treino

Objetivos

- Aprimorar a leitura de código;
- Compreender o processo de repetição;
- Prever o comportamento da máquina;
- Compreender comando de incrementação;
- Compreender comando de decrementação;
- Contagem crescente;
- Contagem decrescente;

```
01
02
03
04
05
06
07
08 // Comando While
09
10 int contador = 1;
11 while (contador <= 10)
12 {
13     System.out.println(contador);
14     contador++;
15 }
16
17
18
19
20
21
22 // Comando FOR
23
24 for (int contador = 1; contador <= 10; contador++)
25 {
26     System.out.println(contador);
27 }
28
29
30
31
32
33
34
35
```



Roteiro de Treino

Objetivos

- Aprimorar a leitura de código;
- Compreender o processo de repetição;
- Prever o comportamento da máquina;
- Compreender comando de incrementação;
- Compreender comando de decrementação;
- Contagem crescente;
- Contagem decrescente;

```
01
02
03
04
05 // Treino 01
06 for (int contador = 2; contador <= 8; contador++)
07 {
08     System.out.println(contador);
09 }
10
11
12
13 // Treino 02
14 for (int contador = 5; contador <= 10; contador+=2)
15 {
16     System.out.println(contador);
17 }
18
19
20
21 // Treino 03
22 for (int contador = 7; contador < 10; contador++)
23 {
24     System.out.println(contador);
25 }
26
27
28
29 // Treino 04
30 for (int contador = 3; contador < 3; contador++)
31 {
32     System.out.println(contador);
33 }
34
35
```



Roteiro de Treino

Objetivos

- Aprimorar a leitura de código;
- Compreender o processo de repetição;
- Prever o comportamento da máquina;
- Compreender comando de incrementação;
- Compreender comando de decrementação;
- Contagem crescente;
- Contagem decrescente;

```
01
02
03
04
05 // Treino 05
06 for (int contador = 5; contador >= 1; contador--)
07 {
08     System.out.println(contador);
09 }
10
11
12
13 // Treino 06
14 for (int contador = 4; contador >= -2; contador-=2)
15 {
16     System.out.println(contador);
17 }
18
19
20
21 // Treino 07
22 for (int contador = 3; contador > 2; contador--)
23 {
24     System.out.println(contador);
25 }
26
27
28
29 // Treino 08
30 for (int contador = 3; contador > 3; contador--)
31 {
32     System.out.println(contador);
33 }
34
35
```



Roteiro de Treino

Objetivos

- Relembrar o uso de funções;
- Utilizar repetições em função;
- Introduzir a ideia de funções sem retorno
- Exemplos

// Treino 09

```
public void contarAte(int limite)
{
    for (int cont = 0; cont <= limite; cont++)
    {
        System.out.println(cont);
    }
}
```

// Treino 10

```
public void contarAte(int limite)
{
    for (int cont = limite; cont >= 0; cont--)
    {
        System.out.println(cont);
    }
}
```



Roteiro de Treino

Objetivos

- Ideia de contagem;
- Ideia de repetição;
- Desenhar formas para representação visual
- Compreender o processo de repetir
- Compreender o processo de repetir processos de repetição

```
01
02
03
04
05
06
07
08 // Repetição de palavras
09 for (int cont = 0; cont <= 5; cont++)
10 {
11     System.out.println("Bruno");
12 }
13
14
15
16 // Linha Vertical
17 for (int cont = 0; cont <= 5; cont++)
18 {
19     System.out.println("* ");
20 }
21
22
23
24 // Linha Horizontal
25 for (int cont = 0; cont <= 5; cont++)
26 {
27     System.out.print("* ");
28 }
29
30
31
32
33
34
35
```



Roteiro de Treino

Objetivos

- Ideia de contagem;
- Ideia de repetição;
- Desenhar formas para representação visual
- Compreender o processo de repetir
- Compreender o processo de repetir processos de repetição

```
// Quadrado
```

```
for (int linha = 0; linha <= 5; linha++)  
{  
    for (int linha = 0; linha <= 5; linha++)  
    {  
        System.out.print("* ");  
    }  
    System.out.println();  
}
```



APROFUNDAMENTO

Comando WHILE



Comando WHILE

O comando WHILE é um dos comandos de repetição que possibilitam que uma sequência de instruções sejam executadas N vezes. Ele está disponível em toda linguagem de programação e é muito utilizado pelos programadores.

GRAMÁTICA E SEMÂNTICA

É composto apenas de um bloco (WHILE), onde semelhante ao comando IF, **recebe uma condição**. Seu comportamento é de repetir as instruções de seu bloco **enquanto a condição for verdadeira**. Ao passo que quando a condição for falsa, a repetição é finalizada.

* É importante perceber que em suas instruções é necessário modificar a(s) variável(eis) que compõem a condição a fim de não resultar em um looping infinito.

```
01      public int somarAte(int numero) ↗ 5
02      {
03          int soma = 0;
04          int contador = 0;
05
06          while (contador <= numero)
07          {
08              soma = soma + contador;
09              contador++;
10          }
11
12          return soma;
13      }
14
15
```

[Condição Loop] contador <= numero	[Volta]	numero	soma	contador
	Antes do Loop	5	0	0
true	Executar Volta 1	5	0	1
true	Executar Volta 2	5	1	2
true	Executar Volta 3	5	3	3
true	Executar Volta 4	5	6	4
true	Executar Volta 5	5	10	5
true	Executar Volta 6	5	15	6
false	Fim do Loop	5	15	6



Comando WHILE

O comando WHILE é um dos comandos de repetição que possibilitam que uma sequência de instruções sejam executadas N vezes. Ele está disponível em toda linguagem de programação e é muito utilizado pelos programadores.

GRAMÁTICA E SEMÂNTICA

É composto apenas de um bloco (WHILE), onde semelhante ao comando IF, **recebe uma condição**. Seu comportamento é de repetir as instruções de seu bloco **enquanto a condição for verdadeira**. Ao passo que quando a condição for falsa, a repetição é finalizada.

* É importante perceber que em suas instruções é necessário modificar a(s) variável(eis) que compõem a condição a fim de não resultar em um looping infinito.

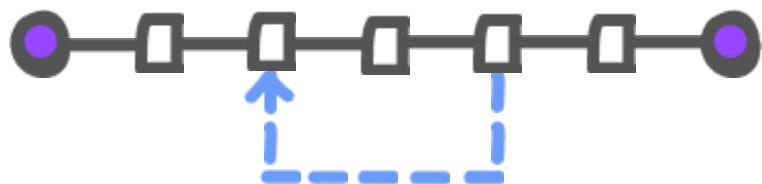
```
01  
02 public int somarParesAte(int numero)  
03 {  
04     int soma = 0;  
05     int contador = 0;  
06  
07     while (contador <= numero)  
08     {  
09         if (contador % 2 == 0)  
10         {  
11             soma = soma + contador;  
12         }  
13         contador++;  
14     }  
15  
16     return soma;  
17 }  
18
```

[Condição Loop] contador <= numero	[Volta]	numero	soma	contador
	Antes do Loop	5	0	0
true	Executar Volta 1	5	0	1
true	Executar Volta 2	5	0	2
true	Executar Volta 3	5	2	3
true	Executar Volta 4	5	2	4
true	Executar Volta 5	5	6	5
true	Executar Volta 6	5	6	6
false	Fim do Loop	5	6	6



Caminho Repetição Simples

O algoritmo que possui caminho de repetição simples é aquele que repete um conjunto de instruções definidas em um bloco N vezes. **Ele é simples por não ter desvios**, mas uma sequência de operações que serão repetidas para resolver o problema.

01234
"Bruno"

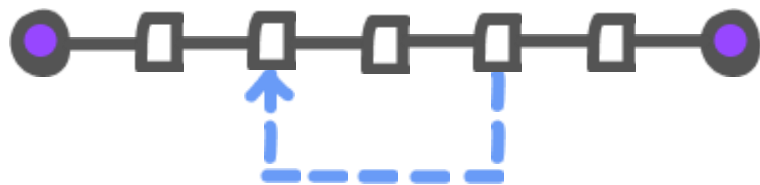
```
01  
02  
03  
04 public string separarLetras(String texto)  
05 {  
06     String novo = "";  
07  
08     int cont = 0;  
09     while (cont < texto.length)  
10     {  
11         char letra = texto.charAt(cont);  
12         novo = novo + letra + " - ";  
13  
14         cont++;  
15     }  
16  
17     return novo;  
18 }  
19  
20
```

[Condição Loop] cont < texto.Length	[Volta]	letra	novo
	Antes do Loop		
true	cont 0	B	""+"B"+"- " => "B- "
true	cont 1	r	"B- "+"r"+"- " => "B-r- "
true	cont 2	u	"B-r-u- "
true	cont 3	n	"B-r-u-n- "
true	cont 4	o	"B-r-u-n-o"
false	Após o Loop		"B-r-u-n-o"



Caminho Repetição Simples

O algoritmo que possui caminho de repetição simples é aquele que repete um conjunto de instruções definidas em um bloco N vezes. **Ele é simples por não ter desvios**, mas uma sequência de operações que serão repetidas para resolver o problema.



```
01  
02  
03  
04 public String inverterTexto(String texto)  
05 {  
06     String textoInvertido = "";  
07     int cont = 0;  
08  
09     while (cont < texto.length)  
10     {  
11         char letra = texto.charAt(cont);  
12         textoInvertido = letra + textoInvertido;  
13  
14         cont++;  
15     }  
16  
17     return textoInvertido;  
18 }  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35
```

01234
"Bruno"

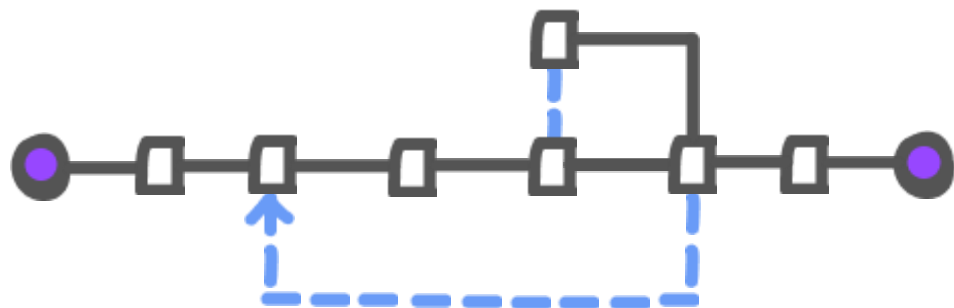
[Condição Loop] cont < texto.Length	[Volta]	letra	textoInvertido
	Antes do Loop		
true	cont 0	B	"B"+" " => "B"
true	cont 1	r	"r"+"B" => "rB"
true	cont 2	u	"u"+"rB" => "urB"
true	cont 3	n	"n"+"urB" => "nurB"
true	cont 4	o	"o"+"nurB" => "onurB"
false	Após o Loop		"onurB"



Caminho

Repetição com Decisão

O algoritmo que possui caminho repetição com decisão é aquele que repete um conjunto de instruções definidas em um bloco N vezes. Ele possui maior complexidade sobre o de repetição simples por **possuir decisões que possibilitam a variação de regra em cada iteração (volta) realizada.**



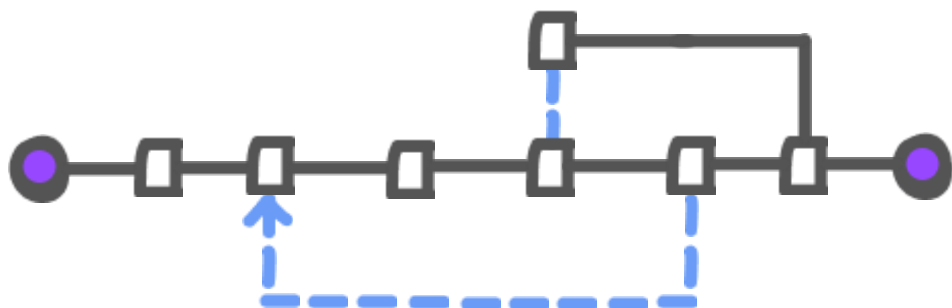
```
01 public int quantasVogais(String texto) 01234
02 {                                     }
03                                     }
04     int qtd = 0;
05     int cont = 0;
06
07     while (cont < texto.length)
08     {
09         char letra = texto.charAt(cont);
10
11         if (letra == 'a' || letra == 'e' || letra == 'i' ||
12             letra == 'o' || letra == 'u')
13         {
14             qtd++;
15         }
16
17         cont++;
18     }
19
20     return qtd;
21 }
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

[Condição Loop] cont < texto.Length	[Volta]	letra	qtd
	Antes do Loop		0
true	cont 0	B	B é vogal? => 0
true	cont 1	r	r é vogal? => 0
true	cont 2	u	u é vogal? => 1
true	cont 3	n	n é vogal? => 1
true	cont 4	o	o é vogal? => 2
false	Após o Loop		2

Caminho

Repetição com Interrupção

O algoritmo que possui caminho repetição com interrupção é aquele que repete um conjunto de instruções definidas em um bloco N vezes, **mas não necessariamente repete N vezes**, isso porque ele possui uma decisão que no caso de ser executada, **interrompe a repetição (mas não a função)**, impedindo que as próximas repetições aconteçam.



```

01
02 public boolean apenasVogais(String texto)
03 {
04     boolean vogais = true;
05     int contador = 0;
06
07     while (contador < texto.length)
08     {
09         char letra = texto.charAt(contador);
10
11         if (letra != 'a' && letra != 'e' && letra != 'i' &&
12             letra != 'o' && letra != 'u')
13         {
14             vogais = false;
15             break;
16         }
17
18         contador++;
19     }
20
21     return vogais;
22 }

```

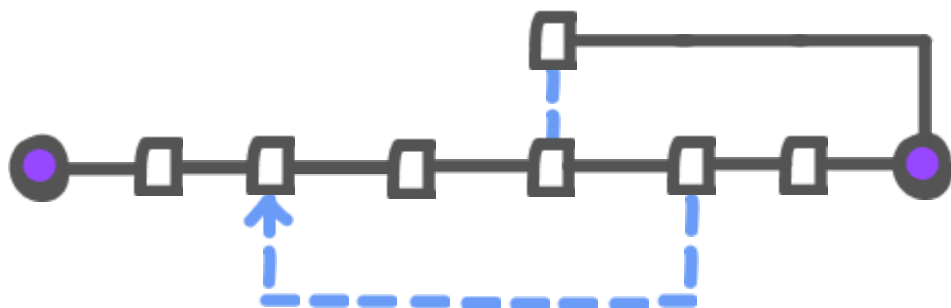
[Condição Loop] cont < texto.Length	[Volta]	letra	vogais
	Antes do Loop		true
true	cont 0	i	i é vogal? => true
true	cont 1	t	r é vogal? => false
interrompida	cont 2	-	-
interrompida	cont 3	-	-
interrompida	cont 4	-	-
-	Após o Loop		false



Caminho Repetição

com Interrupção no algoritmo

O algoritmo que possui caminho repetição com interrupção é aquele que repete um conjunto de instruções definidas em um bloco N vezes, **mas não necessariamente repete N vezes**, isso porque ele possui uma decisão que no caso de ser executada, **interrompe o algoritmo, impedindo qualquer próxima instrução de ser executada**.



```
01 public boolean apenasVogais(String texto)
02 {
03     int contador = 0;
04
05     while (contador < texto.length)
06     {
07         char letra = texto.charAt(contador);
08
09         if (letra != 'a' && letra != 'e' && letra != 'i' &&
10             letra != 'o' && letra != 'u')
11         {
12             return false;
13         }
14
15         contador++;
16     }
17
18     return true;
19 }
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

01234
"italo"

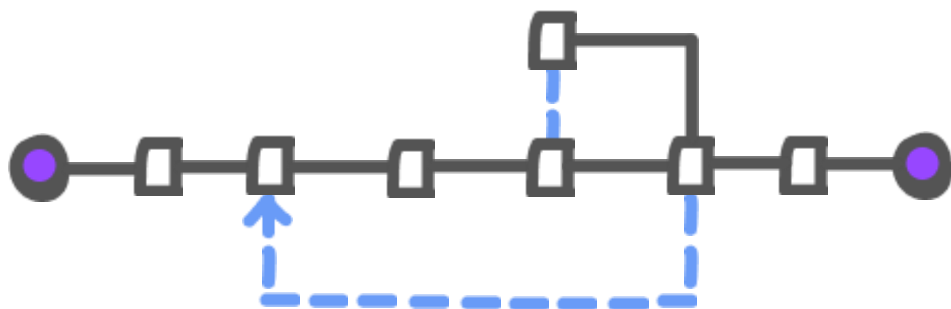
[Condição Loop] cont < texto.Length	[Volta]	letra	[Observação]
	Antes do Loop		
true	cont 0	i	i é vogal
true	cont 1	t	t não é vogal, responde falso
interrompida	cont 2	-	-
interrompida	cont 3	-	-
interrompida	cont 4	-	-
-	Após o Loop		



Caminho Repetição

com Interrupção na volta

O algoritmo que possui caminho repetição com interrupção na volta é aquele que repete um conjunto de instruções definidas em um bloco N vezes, e **pode interromper a volta atual passando para a próxima volta.**



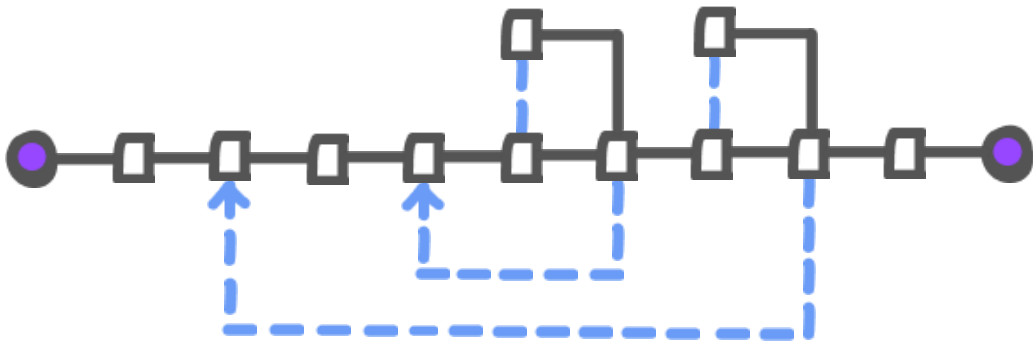
```
01
02
03
04
05
06
07
08
09 public int quantasVogais(String texto)
10 {
11     int qtd = 0;
12     int contador = 0;
13
14     while (contador < texto.length)
15     {
16         char letra = texto.charAt(contador);
17
18         if (letra != 'a' && letra != 'e' && letra != 'i' &&
19             letra != 'o' && letra != 'u')
20         {
21             continue;
22         }
23
24         qtd++;
25         contador++;
26     }
27
28     return qtd;
29 }
30
31
32
33
34
35
```




Caminho Repetição Aninhada

O algoritmo que possui caminho com repetição aninhada é aquele que realiza **repetições dentro de repetições**.

Esse tipo de caminho **é um dos mais complexos** pela dificuldade em visualizar as operações que estão sendo executadas.



```
01
02
03
04
05
06
07 public String letrasUsadas(String texto)
08 {
09     String letrasUsadas = "";
10
11     int i = 0;
12     while (i < texto.length)
13     {
14         boolean letraIncluida = false;
15
16         int j = 0;
17         while (j < letrasUsadas.length)
18         {
19             if (texto.charAt(i) == letrasUsadas.charAt(j))
20                 letraIncluida = true;
21
22             j++;
23         }
24
25         if (!letraIncluida)
26             letrasUsadas = letrasUsadas + texto.charAt(i);
27
28         i++;
29     }
30
31     return letrasUsadas;
32 }
33
34
35
```



APROFUNDAMENTO

Comando FOR

Comando FOR

O comando FOR também é um comandos de repetição e sua característica principal é considerar em sua estrutura uma variável de contador.

GRAMÁTICA E SEMÂNTICA

É composto apenas de um bloco (FOR), onde **recebe três configurações**: (1) O valor inicial do contador de voltas, (2) a condição para que a repetição continue, e (3) o comando que modifica o contador ao fim da volta.

Seu comportamento segue como nos passos abaixo:

1. Inicia o contador;
2. Verifica a condição. Se for verdadeira:
 - 2.1. Executa as instruções do bloco;
 - 2.2. Executa o comando que modifica o contador;
 - 2.3. Volta ao passo 2;
3. Encerra a repetição.

```
01      public int somarAte(int numero)
02      {
03          int soma = 0;
04
05          for (int contador = 0; contador <= numero; contador++)
06          {
07              soma = soma + contador;
08          }
09
10      return soma;
11  }
12
13
```

[Condição Loop] contador <= numero	[Volta]	numero	soma	contador
	Config. do Loop	5	0	0
true	Executar Volta 1	5	0	1
true	Executar Volta 2	5	1	2
true	Executar Volta 3	5	3	3
true	Executar Volta 4	5	6	4
true	Executar Volta 5	5	10	5
true	Executar Volta 6	5	15	6
false	Fim do Loop	5	15	6

Comando FOR

O comando FOR também é um comandos de repetição e sua característica principal é considerar em sua estrutura uma variável de contador.

GRAMÁTICA E SEMÂNTICA

É composto apenas de um bloco (FOR), onde **recebe três configurações**: (1) O valor inicial do contador de voltas, (2) a condição para que a repetição continue, e (3) o comando que modifica o contador ao fim da volta.

Seu comportamento segue como nos passos abaixo:

1. Inicia o contador;
2. Verifica a condição. Se for verdadeira:
 - 2.1. Executa as instruções do bloco;
 - 2.2. Executa o comando que modifica o contador;
 - 2.3. Volta ao passo 2;
3. Encerra a repetição.

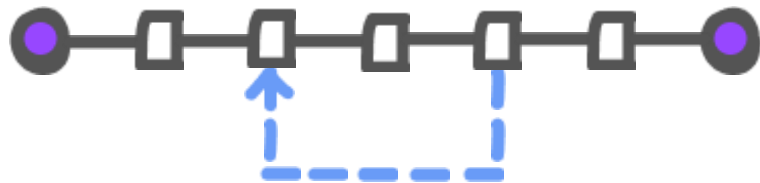
```
01  
02     public int somarParesAte(int numero) ↗ 5  
03     {  
04         int soma = 0;  
05  
06         for (int contador = 0; contador <= numero; contador++)  
07         {  
08             if (contador % 2 == 0)  
09             {  
10                 soma = soma + contador;  
11             }  
12         }  
13  
14         return soma;  
15     }  
16
```

[Condição Loop] contador <= numero	[Volta]	numero	soma	contador
	Config. do Loop	5	0	0
true	Executar Volta 1	5	0	1
true	Executar Volta 2	5	0	2
true	Executar Volta 3	5	2	3
true	Executar Volta 4	5	2	4
true	Executar Volta 5	5	6	5
true	Executar Volta 6	5	6	6
false	Fim do Loop	5	6	6



Caminho Repetição Simples

O algoritmo que possui caminho de repetição simples é aquele que repete um conjunto de instruções definidas em um bloco N vezes. **Ele é simples por não ter desvios**, mas uma sequência de operações que serão repetidas para resolver o problema.

01234
"Bruno"

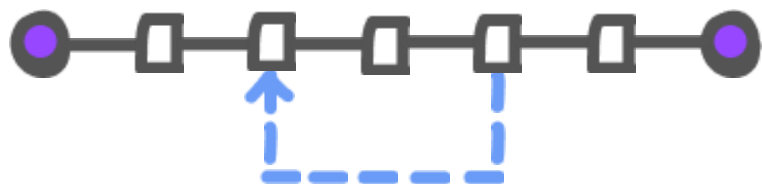
```
01  
02  
03  
04 public String separarLetras(String texto)  
05 {  
06     String novo = "";  
07  
08     for (int contador = 0; contador < texto.length; contador++)  
09     {  
10         char letra = texto.charAt(contador);  
11         novo = novo + letra + "-";  
12     }  
13  
14     return novo;  
15 }  
16  
17  
18  
19  
20  
21
```

[Condição Loop] cont < texto.Length	[Volta]	letra	novo
	Antes do Loop		
true	cont 0	B	""+"B"+"-" => "B-"
true	cont 1	r	"B-"+"r"+"-" => "B-r-"
true	cont 2	u	"B-r-u-"
true	cont 3	n	"B-r-u-n-"
true	cont 4	o	"B-r-u-n-o"
false	Após o Loop		"B-r-u-n-o"



Caminho Repetição Simples

O algoritmo que possui caminho de repetição simples é aquele que repete um conjunto de instruções definidas em um bloco N vezes. **Ele é simples por não ter desvios**, mas uma sequência de operações que serão repetidas para resolver o problema.



```
01  
02  
03  
04 public String inverterTexto(String texto)  
05 {  
06     String textoInvertido = "";  
07     for (int contador = 0; contador < texto.length; contador++)  
08     {  
09         char letra = texto.charAt(contador);  
10         textoInvertido = letra + textoInvertido;  
11     }  
12  
13     return textoInvertido;  
14 }  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35
```

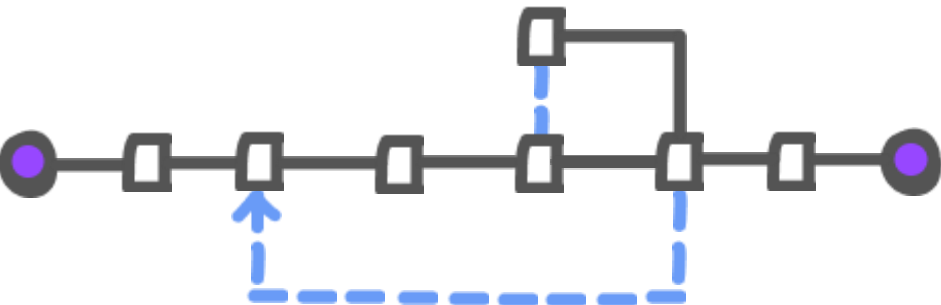
01234
"Bruno"

[Condição Loop] cont < texto.Length	[Volta]	letra	textoInvertido
	Antes do Loop		
true	cont 0	B	"B"+" " => "B"
true	cont 1	r	"r"+"B" => "rB"
true	cont 2	u	"u"+"rB" => "urB"
true	cont 3	n	"n"+"urB" => "nurB"
true	cont 4	o	"o"+"nurB" => "onurB"
false	Após o Loop		"onurB"

Caminho

Repetição com Decisão

O algoritmo que possui caminho repetição com decisão é aquele que repete um conjunto de instruções definidas em um bloco *N* vezes. Ele possui maior complexidade sobre o de repetição simples por **possuir decisões que possibilitam a variação de regra em cada iteração (volta) realizada.**



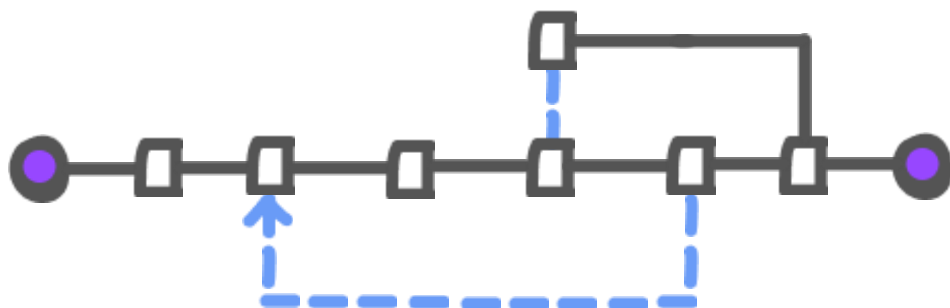
```

01
02 public int quantasVogais(String texto) 01234
03 {                                     }
04     int qtd = 0;
05
06     for (int contador = 0; contador < texto.length; contador++)
07     {
08         char letra = texto.charAt(contador);
09
10         if (letra == 'a' || letra == 'e' || letra == 'i' ||
11             letra == 'o' || letra == 'u')
12         {
13             qtd++;
14         }
15     }
16
17     return qtd;
18 }
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
    
```

[Condição Loop] cont < texto.Length	[Volta]	letra	qtd
	Antes do Loop		0
true	cont 0	B	B é vogal? => 0
true	cont 1	r	r é vogal? => 0
true	cont 2	u	u é vogal? => 1
true	cont 3	n	n é vogal? => 1
true	cont 4	o	o é vogal? => 2
false	Após o Loop		2

Caminho Repetição com Interrupção

O algoritmo que possui caminho repetição com interrupção é aquele que repete um conjunto de instruções definidas em um bloco N vezes, **mas não necessariamente repete N vezes**, isso porque ele possui uma decisão que no caso de ser executada, **interrompe a repetição (mas não a função)**, impedindo que as próximas repetições aconteçam.



```

01
02 public boolean apenasVogais(String texto)
03 {
04     boolean vogais = true;
05
06     for (int contador = 0; contador < texto.length; contador++)
07     {
08         char letra = texto.charAt(contador);
09
10         if (letra != 'a' && letra != 'e' && letra != 'i' &&
11             letra != 'o' && letra != 'u')
12         {
13             vogais = false;
14             break;
15         }
16     }
17
18     return vogais;
19 }

```

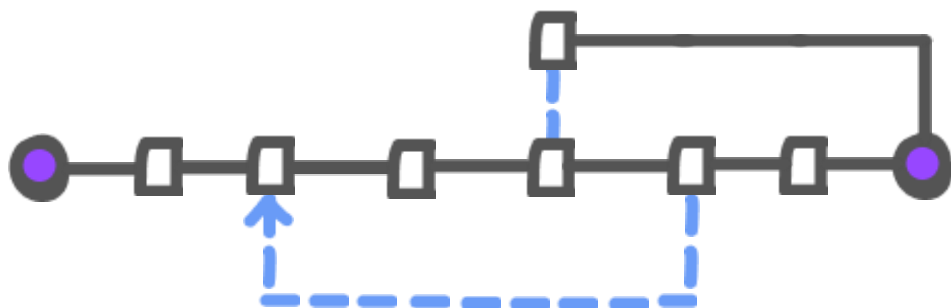
[Condição Loop] cont < texto.Length	[Volta]	letra	vogais
	Antes do Loop		true
true	cont 0	i	i é vogal? => true
true	cont 1	t	r é vogal? => false
interrompida	cont 2	-	-
interrompida	cont 3	-	-
interrompida	cont 4	-	-
-	Após o Loop		false



Caminho Repetição

com Interrupção no algoritmo

O algoritmo que possui caminho repetição com interrupção é aquele que repete um conjunto de instruções definidas em um bloco N vezes, **mas não necessariamente repete N vezes**, isso porque ele possui uma decisão que no caso de ser executada, **interrompe o algoritmo, impedindo qualquer próxima instrução de ser executada**.



```
01 public boolean apenasVogais(String texto)
02 {
03     // 01234
04     // "italo"
05     for (int contador = 0; contador < texto.length; contador++)
06     {
07         char letra = texto.charAt(contador);
08
09         if (letra != 'a' && letra != 'e' && letra != 'i' &&
10             letra != 'o' && letra != 'u')
11         {
12             return false;
13         }
14     }
15     return true;
16 }
17
```

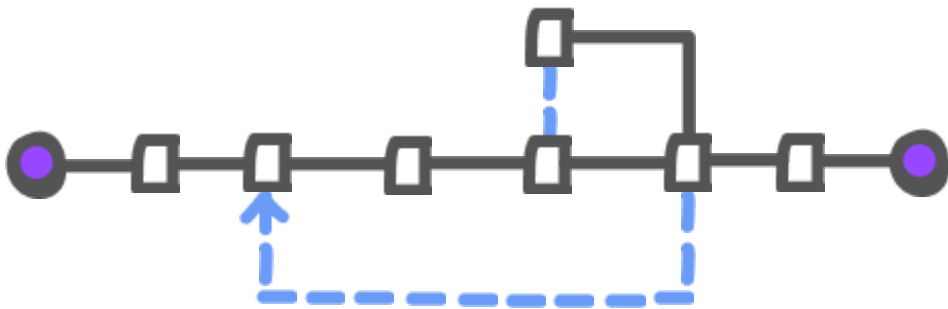
[Condição Loop] cont < texto.Length	[Volta]	letra	[Observação]
	Antes do Loop		
true	cont 0	i	i é vogal
true	cont 1	t	t não é vogal, responde falso
interrompida	cont 2	-	-
interrompida	cont 3	-	-
interrompida	cont 4	-	-
-	Após o Loop		



Caminho Repetição

com Interrupção na volta

O algoritmo que possui caminho repetição com interrupção na volta é aquele que repete um conjunto de instruções definidas em um bloco N vezes, e **pode interromper a volta atual passando para a próxima volta.**



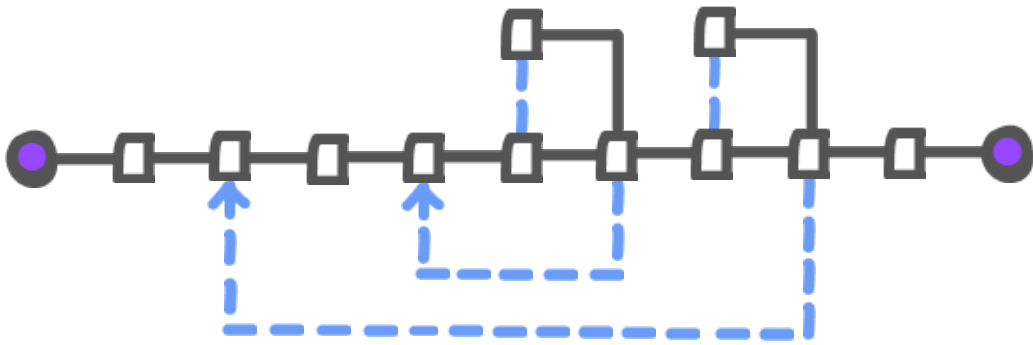
```
01
02
03
04
05
06
07
08
09
10 public int quantasVogais(String texto)
11 {
12     int qtd = 0;
13
14     for (int contador = 0; contador < texto.length; contador++)
15     {
16         char letra = texto.charAt(contador);
17
18         if (letra != 'a' && letra != 'e' && letra != 'i' &&
19             letra != 'o' && letra != 'u')
20         {
21             continue;
22         }
23
24         qtd++;
25     }
26
27     return qtd;
28 }
29
30
31
32
33
34
35
```



Caminho Repetição Aninhada

O algoritmo que possui caminho com repetição aninhada é aquele que realiza **repetições dentro de repetições**.

Esse tipo de caminho **é um dos mais complexos** pela dificuldade em visualizar as operações que estão sendo executadas.



```
01
02
03
04
05
06
07
08 public String letrasUsadas(String texto)
09 {
10     String letrasUsadas = "";
11
12     for (int i = 0; i < texto.length; i++)
13     {
14         boolean letraIncluida = false;
15         for (int j = 0; j < letrasUsadas.length; j++)
16         {
17             if (texto.charAt(i) == letrasUsadas.charAt(j))
18             {
19                 letraIncluida = true;
20             }
21         }
22
23         if (!letraIncluida)
24         {
25             letrasUsadas = letrasUsadas + texto.charAt(i);
26         }
27     }
28
29     return letrasUsadas;
30 }
31
32
33
34
35
```



REPETIÇÃO

Comando FOREACH



Comando FOREACH

Só no próximo encontro ;)

```

90
91 public int SumOfWays(int[] items)
92 {
93     int ways = 0;
94
95     foreach (int item in items)
96     {
97         ways = ways * 2;
98     }
99
100     return ways;
101 }

```

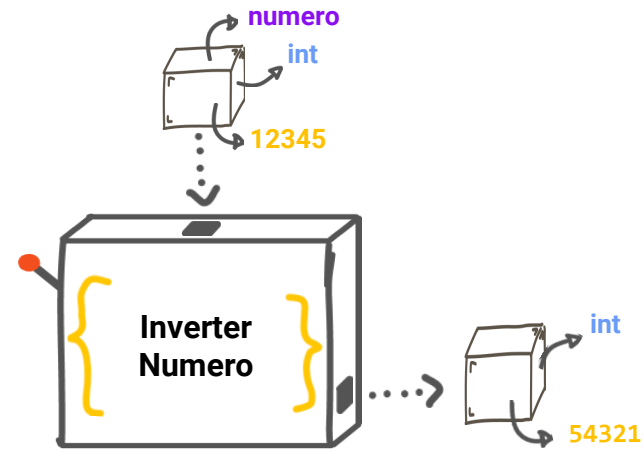


REPETIÇÃO

Treino Prático



Máquina de Função (Inverter Número) em linguagens estáticas

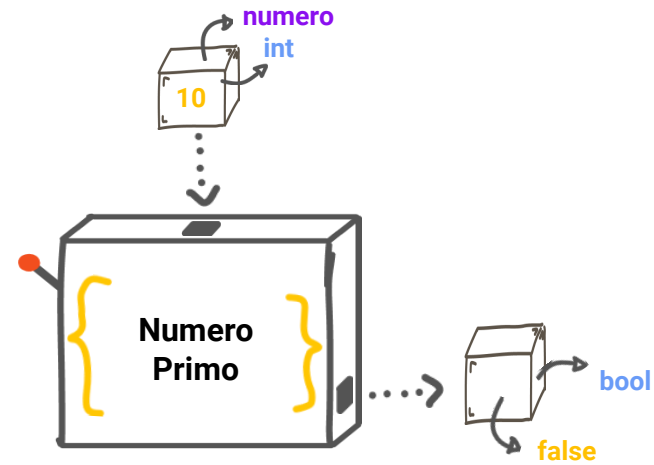


```
01 public int inverterNumero(Integer numero)
02 {
03     String texto = numero.ToString();
04     String novo = "";
05     for (int cont = 0; cont < texto.length; cont++)
06     {
07         novo = texto.charAt(cont) + novo;
08     }
09     int invertido = Integer.parseInt(novo);
10     return invertido;
11 }
12
13
14
15
16
17
18
19
20
21
22
23
24
```

```
25 import java.util.Scanner;
26
27 public void Main(String[] args)
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa INVERTER NÚMERO ##");
32
33         System.out.println("Informe um número:");
34         int a = input.nextInt();
35
36         int x = inverterNumero(a);
37
38         System.out.println(x);
39     }
40     catch (Exception ex) {
41         System.out.println("Ops, ocorreu um erro:");
42         System.out.println(ex.getMessage());
43     }
44 }
45
46
47
48
```



Máquina de Função (Numero Primo) em linguagens estáticas



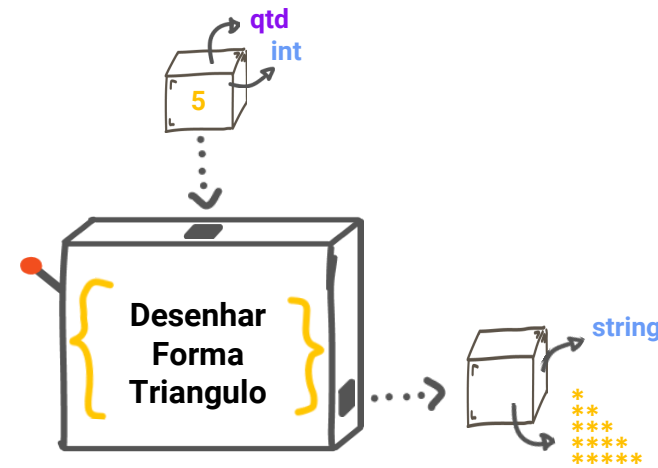
```
01 public boolean numeroPrimo(int numero)
02 {
03     int qtdMultiplo = 0;
04     for (int contador = numero; contador > 0; contador--)
05     {
06         if (numero % contador == 0)
07         {
08             qtdMultiplo++;
09         }
10         if (qtdMultiplo > 2)
11         {
12             return false;
13         }
14     }
15     return true;
16 }
```

```
25 import java.util.Scanner;
26
27 public void Main(String[] args)
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa NUMERO PRIMO ##");
32
33         System.out.println("Informe um número:");
34         int a = input.nextInt();
35
36         boolean x = numeroPrimo(a);
37
38         System.out.println("É primo? " + x);
39     }
40     catch (Exception ex) {
41         System.out.println("Ops, ocorreu um erro:");
42         System.out.println(ex.getMessage());
43     }
44 }
```




Máquina de Função (Desenhar Forma Triângulo)

em linguagens estáticas



```
01 public String desenharFormaTriangulo(int qtd)
02 {
03     String forma = "";
04     for (int linha = 1; linha <= qtd; linha++)
05     {
06         for (int coluna = 1; coluna <= linha; coluna++)
07         {
08             forma = forma + "*";
09         }
10         forma = forma + "\n";
11     }
12     return forma;
13 }
14
15
16 }
```

```
25 import java.util.Scanner;
26
27 public void Main()
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa DESENHAR FORMA ##");
32
33         System.out.println("Informe uma qtd:");
34         int a = input.nextInt();
35
36         String x = desenharFormaQuadrado(a);
37
38         System.out.println(x);
39     }
40     catch(Exception ex) {
41         System.out.println("Ops, ocorreu um erro:");
42         System.out.println(ex.getMessage());
43     }
44 }
45
46 Main();
47
48
```



ENCONTRO 10

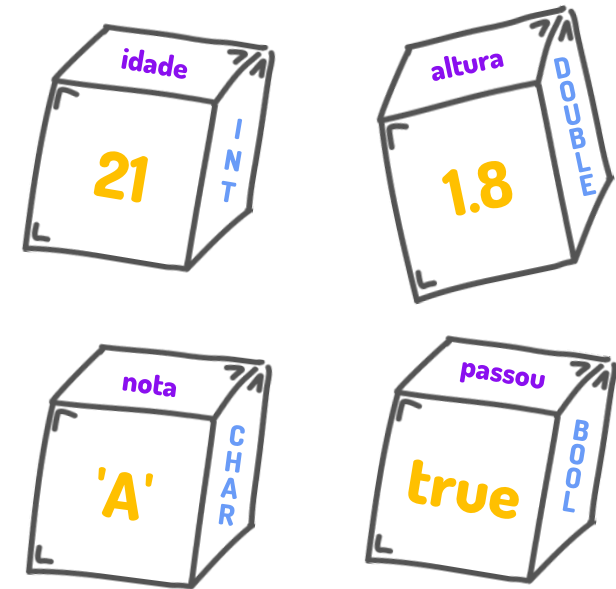
Estruturas de Dados, pt. I



Estruturas Primitivas

Até esse momento, com exceção do tipo *string*, trabalhamos com valores considerados primitivos, ou seja, **valores que não podemos ver do que são formados**, eles são a *matéria prima* para a construção de qualquer outro valor mais complexo. Os valores primitivos que vimos foram números **inteiros**, **decimais**, **caracteres** e **booleanos**.

Por isso, nossas variáveis também guardavam valores primitivos, em outras palavras, elas **guardavam apenas um valor**, e através de um nome (identificador) atribuído a ela, era possível **ler e alterar seu valor**.

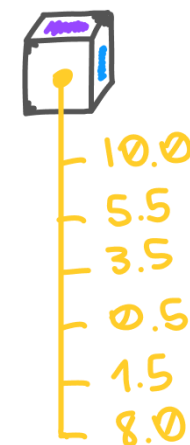
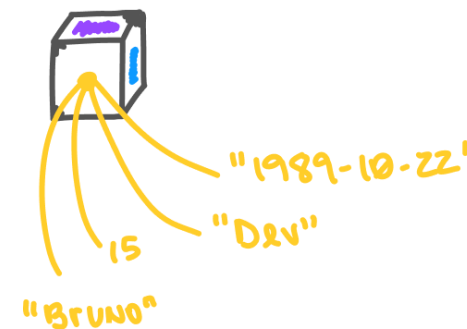
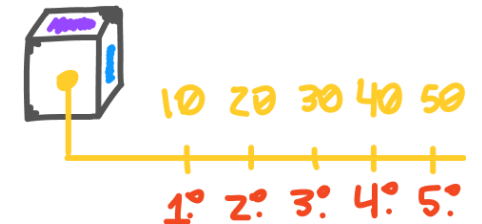
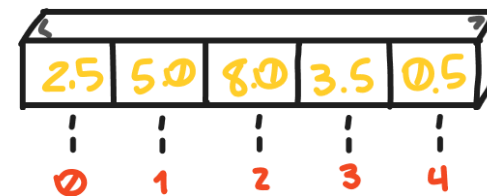




Estruturas Compostas

Conforme o programa ganha funcionalidades que manipulam **dados cada vez mais complexos**, surge a necessidade de criar valores que representem esses dados oferecendo abstrações mais ricas ao programa. Através de estruturas compostas, também chamadas de *estruturas de dados* ou *tipos compostos*, **podemos representar valores que são formados de outros valores**, em outras palavras, **uma variável poderá guardar mais de um valor**.

Toda linguagem de programação oferece meios para criar essas estrutura, que geralmente são entendidas pelos conceitos de **arrays**, **objetos** e **estruturas de dados**.





Principais Estruturas de Dados

A representação de dados complexos é realizada através de estruturas de dados, no entanto é necessário acrescentar que **existem diversas estruturas já conhecidas** no campo da computação, onde cada uma, atua na resolução de um **problema particular**. Cada situação exige uma estrutura que traga maior eficiência e facilidade na manipulação dos dados. As principais estruturas de dados são:

- **Array, Matriz**
- **List** (*Lista*)
- **Linked List** (*Lista ligada*)
- **Queue** (*Fila*)
- **Stack** (*Pilha*)
- **Dictionary** (*Dicionário*)
- **Set** (*Conjunto*)



Manipulando valores de Estruturas Compostas

Estruturas compostas agrupam um conjunto de valores e atribuem à uma única variável.

Qual é o benefício de agrupar valores? Por que não criamos as variáveis de forma independente?

Representar ideias complexas requer a criação de nomes que referenciem estruturas complexas com uma **organização e sistema de regras particular**. À esse processo chamaremos de **abstração**.

Podemos criar nossos próprios tipos compostos ou usar os que a linguagem disponibiliza. Independente da escolha, **é imperioso conhecer como seus elementos são organizados e quais operações/regras podemos realizar em sua manipulação**.



Estruturas de Dados

Array

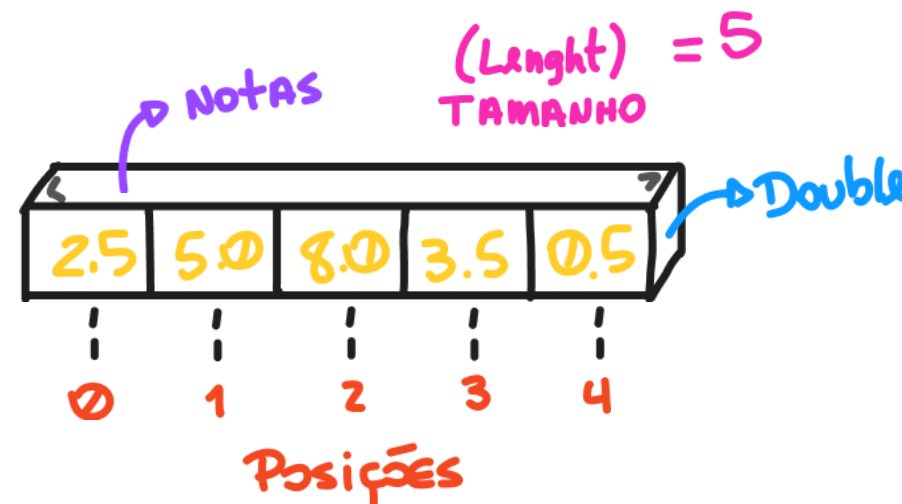


Array

Array é um tipo de estrutura composta, ou seja, é um tipo que **agrupa um conjunto de valores em uma única variável**. Esse conjunto de valores normalmente pertencem a um único tipo.

Seu uso objetiva facilitar a manipulação de grandes volumes de dados que não podem ser previstos anteriormente pelo programador. Dessa forma, ao invés do programador criar 100 variáveis, ele pode criar um Array que guardará um conjunto de 100 valores.

Arrays organizam seus elementos de forma **ordenada** através de **posições fixas iniciadas em 0**, usadas para encontrar seus valores. Depois de definido o tamanho do Array, ou seja, a quantidade de elementos que ele pode guardar, **não se pode mais adicionar nem remover elementos, apenas alterar e ler os valores de cada posição**.





Declaração de Array

Vimos que para criar **variáveis de tipos primitivos**, definimos o tipo da variável, um identificador e um valor inicial **[05:08]**.

Para criar **variáveis de tipos compostos**, especificamente **Arrays**, definimos o tipo seguido de colchetes, um identificador e o operador **new** para inicializar a variável. É possível inicializar Arrays de diversas formas **[15:31]**.

Perceba que o operador **new** é responsável por criar os valores compostos e alocá-los na **Memória RAM**.

```
01
02
03
04
05 // Variável primitiva
06 int numero = 4;
07 boolean passou = false;
08 char nota = 'A';
09
10
11
12
13
14 // Array: Opção 1
15 int[] numeros;
16 numeros = new int[4];
17
18
19
20 // Array: Opção 2
21 int[] numeros = new int[] { 1, 10, 5, 0 };
22
23
24
25 // Array: Opção 3
26 int[] numeros = { 1, 10, 5, 0 };
27
28
29
30
31
32
33
34
35
```



Operações com Array

Após criar um Array [05], podemos realizar as seguintes operações:

1. Ler o valor de uma posição [10];
2. Alterar o valor de uma posição [19];
3. Ler a quantidade de elementos de seu conjunto [29];

A posição é sempre informada dentro do colchete. Quando lemos o valor de um Array, **uma cópia é feita** para o variável que recebe o valor. Ao alterar o valor de um Array, **o valor antigo é substituído** por um novo.

```
01
02
03
04
05  int[] numeros = new int[] { 1, 10, 5, 0 };
06
07
08
09
10  int a = numeros[0];
11
12  System.out.println(a);
13  System.out.println(numeros[0]);
14
15
16
17
18
19  numeros[0] = 2;
20
21  System.out.println(a);
22  System.out.println(numeros[0]);
23
24
25
26
27
28
29  int b = numeros.length;
30
31  System.out.println(b);
32  System.out.println(numeros.length);
33
34
35
```



Operações com Array

Assim como em operações com variáveis simples, que **extraem os valores das variáveis antes de realizar uma operação/função**, também é possível realizar operações com Arrays sem que seja necessário ler o valor em uma variável [17], assim como é possível informar a posição do Array por variáveis [31] ou pelo resultado de uma operação, desde que retorne um número inteiro [32].

```
01
02
03
04
05  int[] numeros = new int[] { 1, 10, 5, 0 };
06
07
08
09  // Opção 1
10  int a = numeros[0];
11  int b = numeros[1];
12  int c = a + b;
13
14
15
16  // Opção 2
17  int d = numeros[0] + numeros[1];
18
19
20
21  // Opção 3
22  int e = 0;
23  int f = 1;
24
25  int g = numeros[e] + numeros[f];
26  int h = numeros[e+1] + numeros[f-1];
27
28
29
30  // Opção 4
31  numeros[e] = 10 + 5;
32  numeros[e+1] = a + b;
33
34
35
```



Iterando em Array

Iterar é sinônimo de fazer novamente. Iterar sob um array é o mesmo que dizer que será realizada **uma repetição lendo cada valor que compõe o array**.

É possível realizar essa tarefa pelos comandos de repetição já vistos, **while** e **for**. Para isso, usaremos o contador **iniciando em 0** e finalizando no tamanho do array. Nesse caso, o contador é usado para ler a posição de cada elemento.

O comando **foreach** é o mais indicado para iterar sob estruturas recursivas, já que ele **lê cada item do array e automaticamente atribui à uma variável a cada volta**.

```
01
02
03
04
05  int[] numeros = new int[] { 1, 10, 5, 0 };
06
07
08
09
10  int cont = 0;
11  while (cont < numeros.length)
12  {
13      System.out.println(numeros[cont]);
14      cont++;
15  }
16
17
18
19
20  for (int i = 0; i < numeros.length; i++)
21  {
22      System.out.println(numeros[i]);
23  }
24
25
26
27
28  for (int item : numeros)
29  {
30      System.out.println(item);
31  }
32
33
34
35
```



Outros tipos Array

Arrays podem ser criados para qualquer outro tipo, primitivo ou não. É possível até mesmo, criar array de arrays **[26:30]**, chamados *jagged arrays*.

Permanecem as mesmas regras vistas para qualquer tipo que seja o array, isto é, pode-se ler e alterar seus elementos por uma posição e descobrir o tamanho de elementos.

```
01
02
03
04
05
06
07
08
09
10
11  int[] numeros = new int[] { 1, 10, 5, 0 };
12
13
14
15  String[] nomes = new String[] { "Bruno", "Junior", "Luiza" };
16
17
18
19  double[] notas = new double[] { 5.5, 8.0, 7.5, 3.5 };
20
21
22
23  boolean[] alternativas = new boolean[] { true, false, false };
24
25
26  int[][] numeros = new int[][]
27  {
28      new int[] { 1, 2, 3 },
29      new int[] { 4, 5, 6 }
30  };
31
32
33
34
35
```



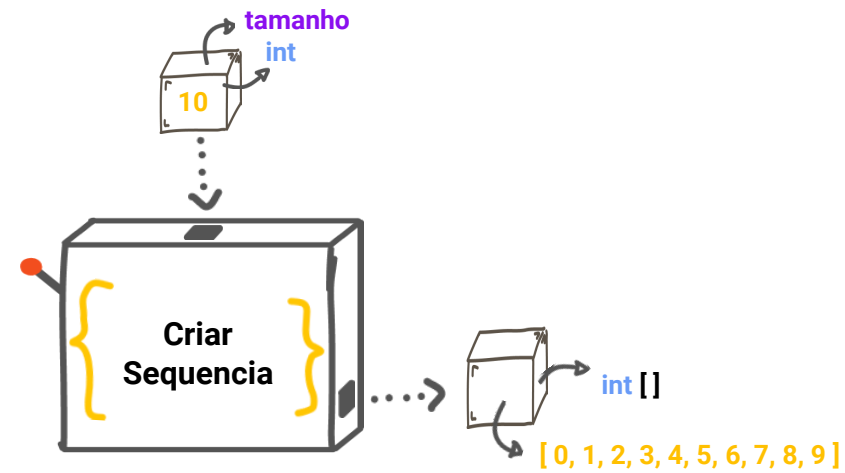
Array

Treino Prático



Máquina de Função (Criar Sequência)

em linguagens estáticas



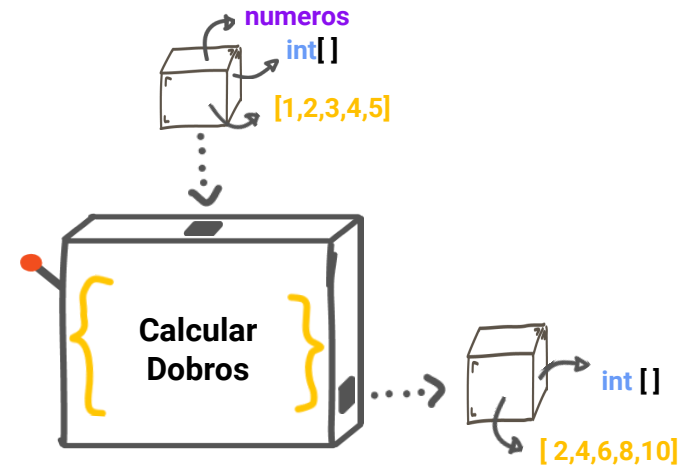
```
01
02 public int[] criarSequencia(int tamanho)
03 {
04     int[] sequencia = new int[tamanho];
05
06     for (int i = 0; i < tamanho; i++)
07     {
08         sequencia[i] = i;
09     }
10
11     return sequencia;
12 }
13
14
15
16 public void printarArray(int[] array)
17 {
18     for (int item : array)
19     {
20         System.out.println(item);
21     }
22 }
23
24
```

```
25
26 import java.util.Scanner;
27
28 public void Main(String[] args)
29 {
30     try {
31         Scanner input = new Scanner(System.in);
32         System.out.println("## Programa CRIAR SEQUENCIA ##");
33
34         System.out.println("Informe um número:");
35         int a = input.nextInt();
36
37         int[] x = criarSequencia(a);
38
39         printarArray(x);
40     }
41     catch (Exception ex) {
42         System.out.println("Ops, ocorreu um erro:");
43         System.out.println(ex.getMessage());
44     }
45 }
46
47
48
```



Máquina de Função (Calcular Dobros)

em linguagens estáticas

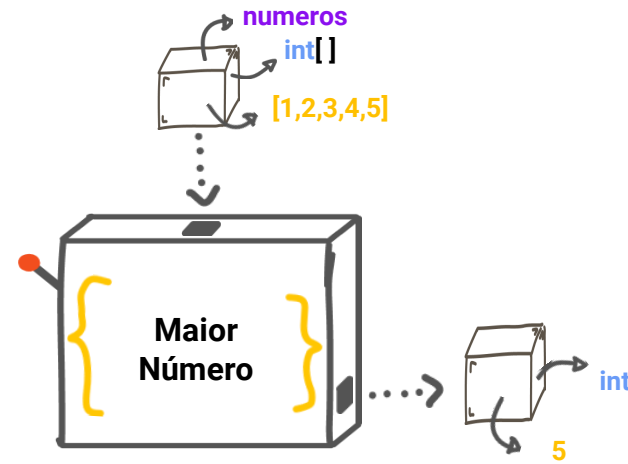


```
01
02 public int[] CalcularDobros(int[] numeros)
03 {
04     int[] dobros = new int[numeros.length];
05
06     for (int i = 0; i < dobros.length; i++)
07     {
08         dobros[i] = numeros[i] * 2;
09     }
10
11     return dobros;
12 }
13
14
15
16 public void PrintarArray(int[] array)
17 {
18     for (int item : array)
19     {
20         System.out.println(item);
21     }
22 }
23
24
```

```
25 import java.util.Scanner;
26
27 public void Main(String[] args)
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa DOBROS ##");
32
33         System.out.println("Informe 5 números de cada vez:");
34
35         int[] a = new int[5];
36         for (int i = 0; i < 5; i++)
37             a[i] = input.nextInt();
38
39         int[] x = CalcularDobros(a);
40
41         PrintarArray(x);
42     }
43     catch (Exception ex) {
44         System.out.println("Ops, ocorreu um erro:");
45         System.out.println(ex.getMessage());
46     }
47 }
48
```




Máquina de Função (Maior Número) em linguagens estáticas



```
01 public int maiorNumero(int[] numeros)
02 {
03     int maior = 0;
04
05     for (int item : numeros)
06     {
07         if (item > maior)
08             maior = item;
09     }
10
11     return maior;
12 }
13
14
15
16
17
18
19
20
21
22
23
24
```

```
25 import java.util.Scanner;
26
27 public void Main(String[] args)
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa MAIOR NÚMERO ##");
32
33         System.out.println("Informe 5 números de cada vez:");
34
35         int[] a = new int[5];
36         for (int i = 0; i < 5; i++)
37             a[i] = input.nextInt();
38
39         int x = maiorNumero(a);
40
41         System.out.println("O maior número é " + x);
42     }
43     catch (Exception ex) {
44         System.out.println("Ops, ocorreu um erro:");
45         System.out.println(ex.getMessage());
46     }
47 }
48
```



Estruturas de Dados

Matriz



Matriz

Uma Matriz é semelhante a um Array, com a diferença de ter mais de uma dimensão, enquanto um Array armazena uma sequência de valores ordenados, uma Matriz bidimensional armazena seus valores em formato de uma tabela, através da ideia de linhas e colunas

Matrizes são usadas justamente quando a estrutura dos dados não representam uma sequência única de valores, mas quando existe relação entre as dimensões do conjunto de dados. Podemos pensar uma Matriz como um conjunto de Arrays de mesmo tamanho.

Assim como Arrays, depois de definido o tamanho das dimensões, **não se pode mais adicionar nem remover elementos, apenas alterar e ler os valores de cada posição.**



Declaração de Matriz

Para criar *Arrays*, definimos o tipo seguido de colchetes, um identificador e iniciamos a variável **[03]**.

Para criar uma Matriz, definimos o tipo, seguido de colchetes, onde cada par de colchetes é uma dimensão. Seguimos com um identificador e iniciamos a Matriz informando o tamanho das dimensões ou mantendo vazio caso já seja iniciada **[10]**, normalmente entendemos o primeiro valor como sendo as linhas e o segundo, as colunas.

Também é possível iniciar uma Matriz sem informar os tamanhos das dimensões, nesse caso, a linguagem irá inferí-los através do valor inicial informado **[18, 28]**.

```
01
02 // Array
03 int[] numeros = new int[] { 1, 10, 5, 0 };
04
05
06
07
08
09 // Matriz 2x2
10 int[][] numeros = new int[][]
11 {
12     {1, 2},
13     {3, 4}
14 };
15
16
17 // Matrix 4x2
18 int[][] numeros = new int[][]
19 {
20     {1, 2},
21     {3, 4},
22     {5, 6},
23     {7, 8}
24 };
25
26
27 // Matrix 2x4
28 int[][] numeros = new int[][]
29 {
30     {1, 2, 3, 4},
31     {5, 6, 7, 8}
32 };
33
34
35
```



Operações com Matriz

Após criar uma Matriz [04-10], podemos realizar as seguintes operações:

1. Ler o valor de uma posição [16];
2. Alterar o valor de uma posição [22];
3. Ler a quantidade de elementos de seu conjunto [28-29];

A posição é sempre informada dentro do colchete. Quando lemos o valor de uma Matriz, **uma cópia é feita** para o variável que recebe o valor. Ao alterar o valor de uma Matriz, **o valor antigo é substituído** por um novo.

```
01
02
03
04  int[][] numeros = new int[][]
05  {
06      {1, 2},
07      {3, 4},
08      {5, 6},
09      {7, 8}
10  };
11
12
13
14
15
16  int a = numeros[0][1];
17  System.out.println(a);
18  System.out.println(numeros[0][1]);
19
20
21
22  numeros[0][1] = 11;
23  System.out.println(a);
24  System.out.println(numeros[0][1]);
25
26
27
28  int b = numeros.length;
29  int c = numeros[1].length;
30  System.out.println(b);
31  System.out.println(c);
32
33
34
35
```



Operações com Matriz

Assim como em operações com variáveis simples, que **extraem os valores das variáveis antes de realizar uma operação/função**, também é possível realizar operações com Matrizes sem que seja necessário ler o valor em uma variável [24], assim como é possível informar a posição do Array por variáveis [32] ou pelo resultado de uma operação, desde que retorne um número inteiro [33].

```
01
02
03
04  int[][] numeros = new int[][]
05  {
06      {1, 2},
07      {3, 4},
08      {5, 6},
09      {7, 8}
10  };
11
12
13
14
15
16  // Opção 1
17  int a = numeros[0][0];
18  int b = numeros[0][1];
19  int c = a + b;
20
21
22
23  // Opção 2
24  int d = numeros[0][0] + numeros[0][1];
25
26
27
28  // Opção 3
29  int e = 0;
30  int f = 1;
31
32  int g = numeros[e][f] + numeros[f][e];
33  int h = numeros[e+2][f-1] + numeros[e+3][f];
34
35
```



Iterando em Matriz

Como vimos, iterar é sinônimo de fazer novamente. Iterar sob uma Matriz é o mesmo que dizer que será realizada **uma repetição lendo cada valor que compõe a Matriz.**

Normalmente utilizamos a **estrutura for de forma aninhada** para iterar sob uma Matriz, onde cada *for* representa uma dimensão. O limite da repetição é dado através do resultado da propriedade *length* que retorna o tamanho da dimensão.

Perceba que a leitura é feita linha a linha, ou seja, lê-se os valores das colunas da primeira linha e em seguida avançamos para a próxima linha. O nome dos contadores indicam a dimensão *linha* e *coluna*.

```
01  
02  
03  
04  
05  
06  
07  
08  
09  int[][] numeros = new int[][]  
10  {  
11      {1, 2},  
12      {3, 4},  
13      {5, 6},  
14      {7, 8}  
15  };  
16  
17  
18  
19  
20  
21  
22  for (int linha = 0; linha < numeros.length; linha++)  
23  {  
24      for (int coluna = 0; coluna < numeros[linha].length; coluna++)  
25      {  
26          System.out.println(numeros[linha][coluna]);  
27      }  
28  }  
29  
30  
31  
32  
33  
34  
35
```



Outros tipos

Matriz

Matrizes podem ser criadas para qualquer outro tipo. Permanecem as mesmas regras vistas para qualquer tipo que seja a Matriz, isto é, pode-se ler e alterar seus elementos informando a posição das dimensões, descobrir o tamanho de cada dimensão e iterar sob seus items.

```
01
02
03
04  int[][] numeros =
05  {
06      { 1, 2 },
07      { 3, 4 }
08  };
09
10
11
12  String[][] familias =
13  {
14      { "Bruno", "Junior", "Luiza" },
15      { "Penelope", "Carlos", "Lucia" }
16  };
17
18
19
20  double[][] notas =
21  {
22      { 5.5, 8.0, 7.5 },
23      { 7.5, 4.0, 6.5 },
24      { 6.5, 9.0, 9.5 }
25  };
26
27
28
29  boolean[][] alternativas =
30  {
31      { true, false, false, false },
32      { false, true, true, true }
33  };
34
35
```



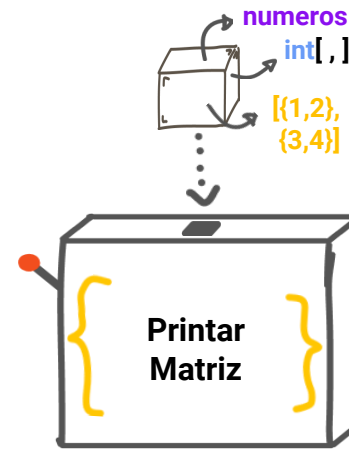

Array

Treino Prático



Máquina de Função (Printar Matriz)

em linguagens estáticas



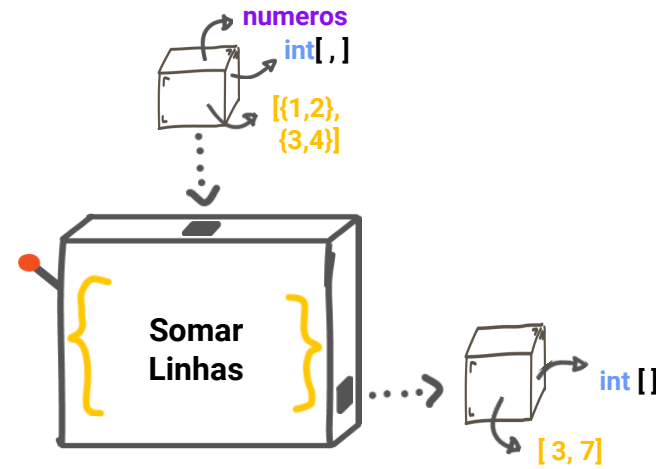
```
01
02
03
04 public void printarMatriz(int[][] numeros)
05 {
06     for (int linha = 0; linha < numeros.length; linha++)
07     {
08         for (int coluna = 0; coluna < numeros[linha].length; coluna++)
09         {
10             System.out.println(numeros[linha][coluna]);
11         }
12     }
13 }
14
15
16
17
18
19
20
21
22
23
24
```

```
25 import java.util.Scanner;
26
27 public void Main(String[] args)
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa PRINTAR MATRIZ ##");
32
33         int[][] a = new int[2][2];
34
35         for (int i = 0; i < 2; i++)
36             for (int j = 0; j < 2; j++)
37                 a[i][j] = input.nextInt();
38
39         printarMatriz(a);
40     }
41     catch (Exception ex) {
42         System.out.println("Ops, ocorreu um erro:");
43         System.out.println(ex.getMessage());
44     }
45 }
46
47
48
```



Máquina de Função (Somar Linhas)

em linguagens estáticas



```
01 public void somarLinhas(int[][] numeros)
02 {
03     int[] seq = new int[numeros.length];
04     for (int linha = 0; linha < numeros.length; linha++)
05     {
06         int soma = 0;
07         for (int coluna = 0; coluna < numeros[linha].length; coluna++)
08         {
09             soma += numeros[linha][coluna];
10         }
11         seq[linha] = soma;
12     }
13 }
14
15
16
17 public void printarArray(int[] array)
18 {
19     for (int i = 0; i < tamanho; i++)
20     {
21         System.out.println(array[i]);
22     }
23 }
24
```

```
25 import java.util.Scanner;
26
27 public void Main()
28 {
29     try {
30         Scanner input = new Scanner(System.in);
31         System.out.println("## Programa PRINTAR MATRIZ ##");
32
33         int[][] a = new int[2][2];
34
35         for (int i = 0; i < 2; i++)
36             for (int j = 0; j < 2; j++)
37                 a[i][j] = input.nextInt();
38
39         int[] x = somarLinhas(a);
40         printarArray(x);
41     }
42     catch (Exception ex) {
43         System.out.println("Ops, ocorreu um erro:");
44         System.out.println(ex.getMessage());
45     }
46 }
47
48 Main();
```



Obrigado!
Dúvidas?

Bruno de Oliveira
brunodeoliveira.22.10@gmail.com