**AULA 1 – PENSAMENTO COMPUTACIONAL**

Processo de pensamento envolvido na expressão de soluções em passos computacionais ou algoritmos que podem ser implementados no computador.

Não se restringe ao meio computacional.

É um pensamento sistemático e eficiente, uma formulação e resolução de problemas. E tanto humanos quanto máquinas devem ser capazes de resolver o pensamento computacional.

É uma habilidade generalista, e não uma disciplina acadêmica.

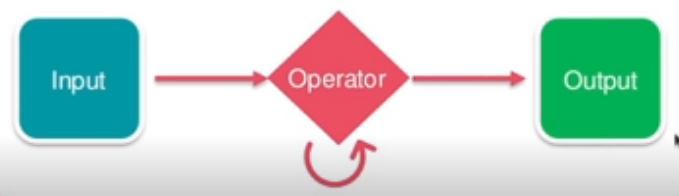
É divido em 4 pilares:

**a) decomposição ->** pegar um problema de um certo nível de complexidade e dividi-lo de uma forma que consiga encontrar problemas menores e resolvidas. Dessa forma, se ataca um ponto de cada vez, e não trabalha com um tema complexo. Ex: bolo é composto por massa, recheio e cobertura. Para fazer, não preciso seguir uma ordem, porém para montagem, sigo a ordem necessária. Pesquisa binária (melhor utilizada do que pesquisa vetorial).

**b) reconhecimento de padrões ->** verificar similaridades, diferenças e tendências dentro de um contexto ou de contextos distintos. As similaridades ou diferenças podem ocorrer dentro de contextos distintos. Também serve para padrão comportamental.

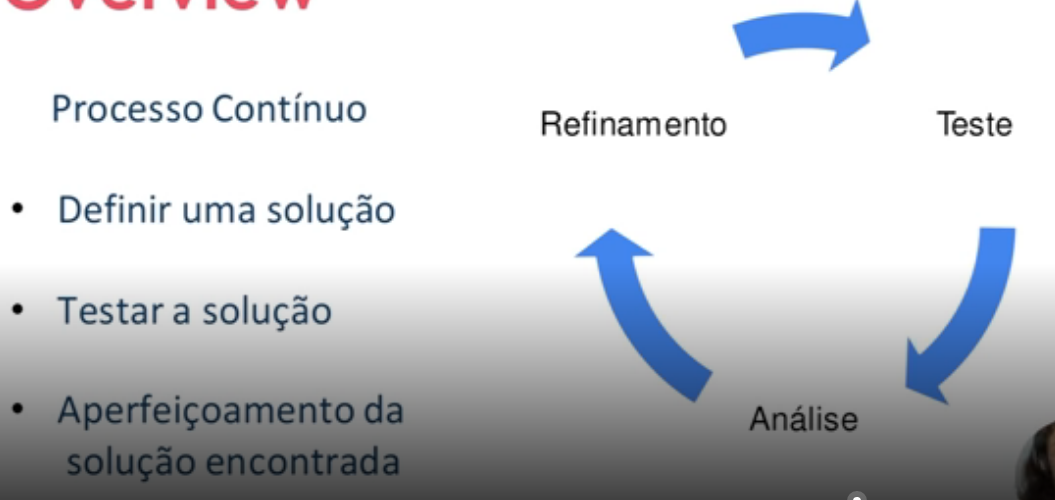
**c) abstração ->** extrapolar o conceito de um problema particular, específico, para uma forma generalista. Ou seja, pego do mundo concreto e levo para o mundo das ideias. Por exemplo: quero saber a circunferência da caixa de bluetooth, só que, se medir a circunferência do círculo em questão, somente identifico a resolução para este problema. Contudo, se eu consigo parametrizar essa informação, consigo generalizá-la, e posso utilizar essa informação para qualquer círculo que exista, como é o caso da fórmula da circunferência (C=2πR). Dessa forma, consigo tirar do particular/específico para o generalista. Ex2: o designer deve pensar como o layout ficará em todas as telas e plataformas. Ex3: as classes são abstrações de um modelo de identidade de software.

**d) design de algoritmos ->** aqui, definimos as instruções de passo a passo da resolução de problemas.



Dado uma entrada (input), tenho uma sucessão de operações (operator), que serão executadas consequêncialmente ou dependendo de determinada condição, e vão gerar um resultado (output). Se esse resultado é esperado ou não, vai depender de como se está formulando o algoritmo.

**Esse processo precisa de constante refinamento, aperfeiçoamento:**



**O pensamento computacional nos possibilita utilizar o melhor dos dois mundos: as habilidades humanas e os recursos computacionais.**



A utilização do pensamento computacional acrescenta competência às pessoas, como: pensamento sistemático, melhora colaboração dentro da equipe (pois consegue estruturar melhor a resolução de seu problema, suas ideias), criatividade e design (está atrelado a como consegue identificar os padrões e soluções, explicar melhor, e acaba sendo mais criativo) e torna-se mais facilitador, uma vez que consegue apresentar a resolução de seu problema de uma maneira mais eficiente e objetivo.

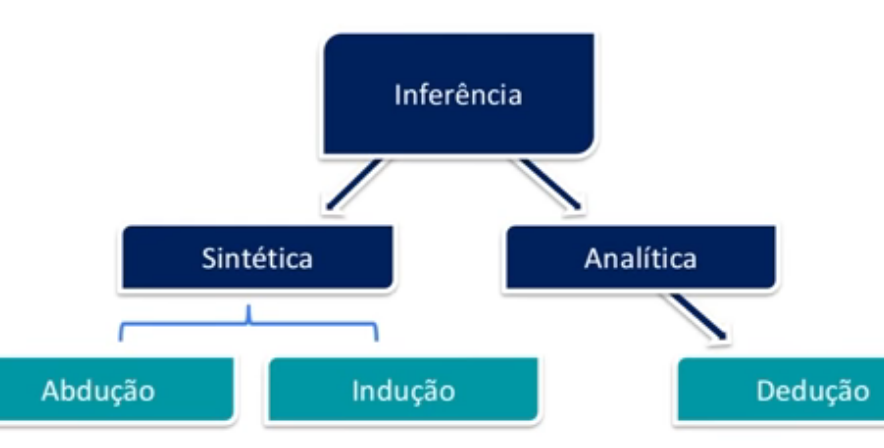
**Habilidades complementares:**

Os tipos de pensamentos que precisamos resolver são: Raciocínio Lógico e Aperfeiçoamento.

**Raciocínio Lógico** é uma forma de pensamento estruturado, ou raciocínio, que permite encontrar a conclusão ou determinar a resolução de um problema. É uma habilidade de treinamento.

Classificação:

1. **Indução ->** geralmente vem a partir de um fenômeno observado e, a partir desse fenômeno, se consegue extrapolar e determinar leis e teorias relacionadas àquele fenômeno. Induz que algo acontece a partir de uma observação. Está muito relacionado a Ciências experimentais/Empirismo.
2. **Dedução ->** trabalho inverso ao da indução. A partir de leis e teorias, você deduz previsões e explicações para aquele fenômeno. É muito utilizado nas ciências exatas.
3. **Abdução ->** a partir de uma conclusão, você tira uma premissa. Ex: a grama está molhada, logo deve ter chovido. Neste caso, não necessariamente a premissa será verdadeira. É muito utilizado em processo investigativo, diagnósticos, em parte de detetives.



**Aperfeiçoamento** a partir de uma solução, determinar pontos de melhora e refinamento, seja de forma pontual ou geral/global. Dentro do ato de aperfeiçoar, temos:

- encontrar solução eficiente ambos significar melhor uso de recursos/otimizar recursos

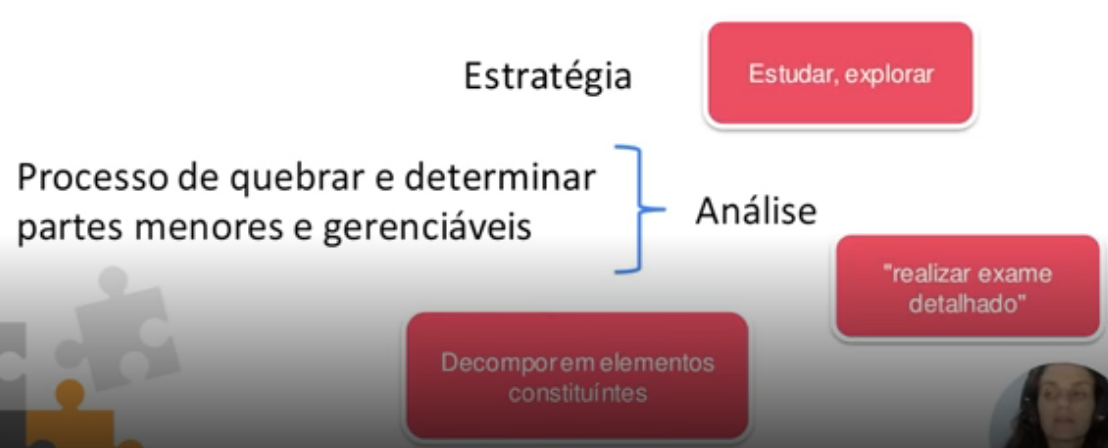
- otimizar processos

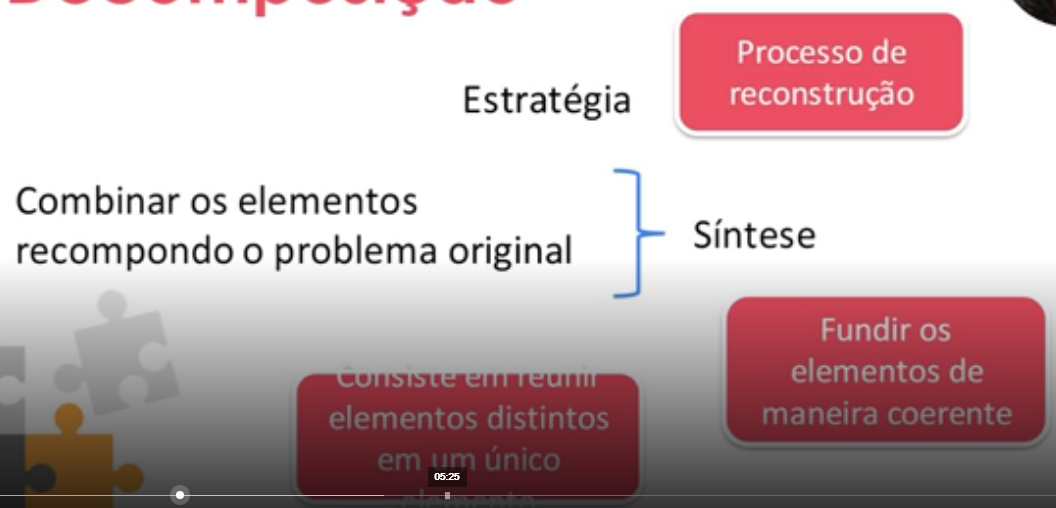
- simplificar linhas de códigos ambos significam melhorar códigos e algoritmos, tornando mais legível

- ter funções bem definidas

**Pilares: Decomposição**

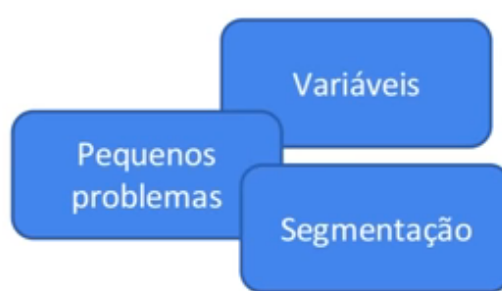
É preciso passo da resolução de problemas dentro do conceito de pensamento computacional: dado um problema complexo, devemos quebrá-lo em problemas menores. Portanto, problemas mais fáceis e gerenciáveis.

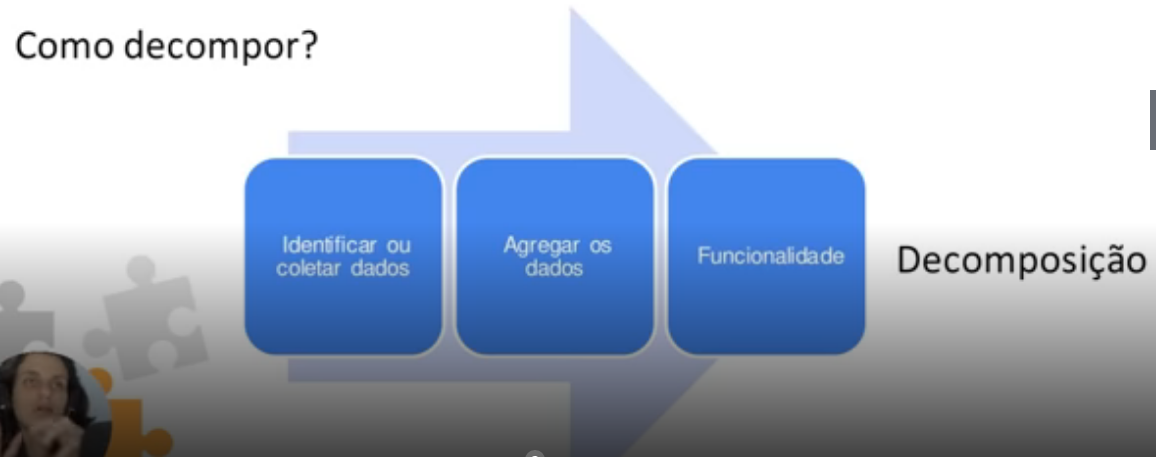




A análise a síntese trabalham juntas.

Ordem da execução das tarefas menores: **sequencial** ou **paralela**. Vai depender do contexto, se há dependências, do que você precisa. Se for **sequencial**, provavelmente existirá uma dependência entre as tarefas, execução em fila. Se as tarefas podem ser executadas concomitantemente, a estratégia usada é a **paralela**, pois é mais eficiente e economiza tempo. As tarefas são isoladas e independentes.





**Pilares: Padrões**

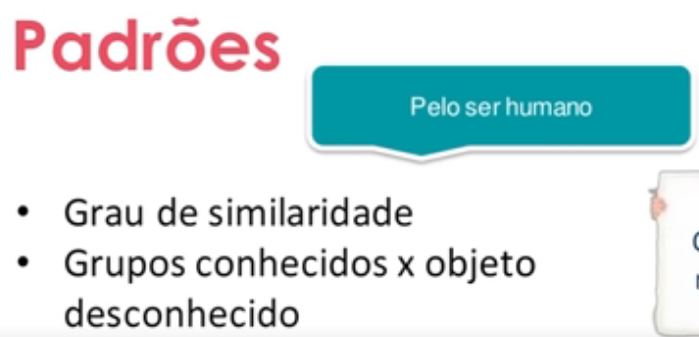
Preciso de um modelo base/de referencia, que determina uma estrutura invariante e repetição. A partir de um modelo, eu posso determinar objetos diferentes em que repita aquela estrutura.

Os padrões são reconhecidos através das similaridades e diferenças.

Ex: compressão de dados; armazenamento de fotos pelos servidores.

Por que determinar padrões? Pela vantagem de generalizar, com o objetivo de obter resolução para problemas diferentes. Ex: encontrar menor caminho entre A e B.





**O computador faz por COMPARAÇÃO.**

O ser humano é subjetivo, consegue identificar algo por subjetividade. O computador é **objetivo e direto**. Se ele não tem a informação, não consegue fazer a comparação. Então preciso representar os atributos para que a máquina consiga aprender através de um conceito associado ao objeto. Então, ele armazena esses dados para consultas posteriores, afim de determinar a qual categoria o objeto-alvo se encaixa. E, a partir daí, existem as regras de decisão.

**Pilares: Abstração**

**Abstrair** é observar um ou mais elementos, avaliando características e propriedades em separado.

**Abstração** é um processo intelectual de isolamento de um objeto da realidade.

**Generalizar** é tornar-se geral, mais amplo, extensão. Abstrair é generalizar.

**Generalização**, na lógica, é a operação intelectual que consiste em reunir uma classe geral, um conjunto de seres ou fenômenos similares.

Como classificar os dados:

1) analisar as características

2) identificar os pontos essenciais e descartar os detalhes

3) generalizar x detalhar

A ideia de **abstrair** um conceito, de generalizá-lo, é torna-lo reutilizável. É torna-lo geral para que possa ser utilizado por cenários distintos, mas com similaridade entre si.

**Algoritmos**

Um computador não opera sozinho. É necessário determinar instruções detalhadas para que ele possa executar a tarefa requerida. Essas instruções detalhadas são utilizadas em processamento dos dados dos programas. O objetivo de um computador é receber, manipular, armazenar e processar dados.

Os programas são constituídos pelas instruções que, por sua vez, possuem passa-a-passo do que o computador precisa realizar.

As ordens devem ser entendidas tanto pelo humano, quanto pela máquina.

Instruções executadas passo a passo para concluir uma tarefa.

Desenvolvimento do Programa:

1. Análise -> estudo e definição dos dados de entrada e saída;
2. Algoritmo -> descreve o problema por meio de ferramentas narrativas, fluxograma ou pseudocódigo;
3. Codificação -> o algoritmo é codificado de acordo com a linguagem de programação escolhida.

Sequência de passos com objetivo definido -> Execução de tarefas específicas -> conjunto de operações que resultam em uma sucessão finita de ações.

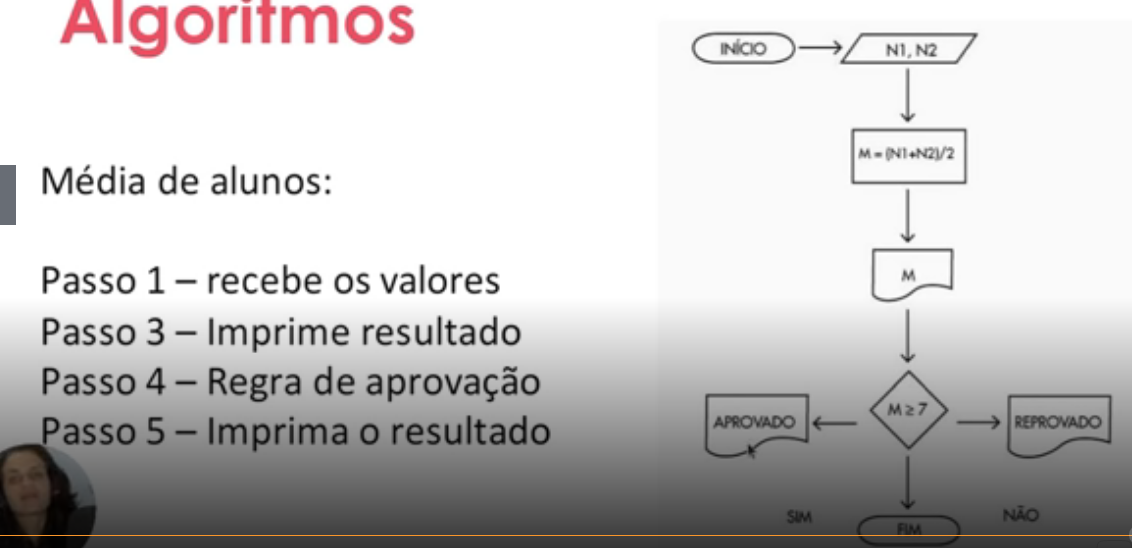
Como construir um algoritmo?

1. Compreensão do problema -> quais pontos mais importantes relacionados ao meu contexto;
2. Definição dos dados de entrada -> definir quais são os dados fornecidos e cenários, que são inerentes ao contexto do problema;
3. Definir processamento -> quais operações que serão realizadas dentro do algoritmo, para que se possa processar os dados e, depois, retornar as informações de saída (cálculos e restrições);
4. Definir dados de saída -> ocorre após o processamento;
5. Utilizar um método de construção -> construção e refinamento do algoritmo;
6. Teste e diagnóstico -> refinamento.

Construção de algoritmo:

1. NARRATIVA -> é a mais simples. Não atribui conceito novos a esse tipo de estruturação de passo a passo (step by step). Utilização de linguagem natural. Pode ser ambígua/diversas interpretações possíveis.
2. FLUXOGRAMA -> é uma estrutura gráfica onde se tem símbolos pré-definidos, que definem qual o tipo de ação que está sendo executada. É de simples entendimento, mas requer o conhecimento prévio da estrutura e dos símbolos utilizados.
3. PSEUDOCÓDIGO -> Portugol. É a parte de regras bem definidas, tem passo a passo a serem executados. É o mais próximo de uma codificação, mas não é uma linguagem de programação efetivamente. É um meio termo para que se possa se acostumar com a parte da programação.

-> NARRATIVA

 -> FLUXOGRAMA

**INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO:**

Lógica é utilizada para solucionar um problema/uma questão, que pode ter numerosas soluções.

**Problema** é uma questão que foge a uma determinada regra, ou melhor, é um desvio de percurso, o qual impede de atingir um objetivo com eficiência ou eficácia.

**Lógica** é uma parte da filosófica que trata das formas do pensamento em geral (dedução, indução, hipótese, inferência etc.) e das operações intelectuais que visam à determinação do que é verdadeiro ou não. É uma organização coesa. Uma forma de raciocínio, de ordenação que segue convenções, uma forma de desencadear acontecimentos.

A **Lógica** é uma organização e planejamento das instruções, assertivas em um algoritmo, a fim de viabilizar a implantação de um programa.

Seres humanos conseguem prever comportamentos, pois utilizamos a lógica de forma intuitiva. Os computadores não. Por isso, as instruções devem ser detalhadas e, para isso, utilizamos a lógica de programação.

**Técnicas de lógica de programação:**

1. Técnica linear:

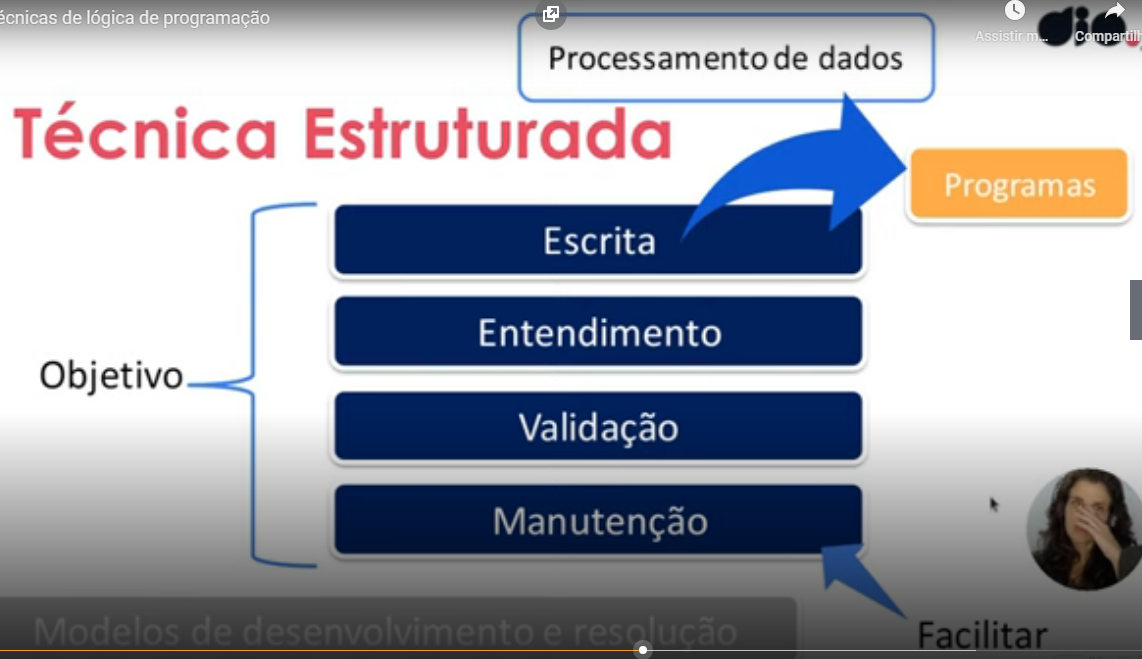
* Modelo tradicional;
* Não tem vínculo;
* Possui estrutura hierárquica;
* É utilizada na programação de computadores.

É uma técnica de modelo de desenvolvimento e resolução de problemas. Bastante associada à matemática.

É a **execução sequenciada** de uma série de operações, p.ex. ordenação de elementos de uma única propriedade, onde temos recursos limitados e uma **única dimensão**. As ações possuem dependência entre si.

1. Técnica estruturada:

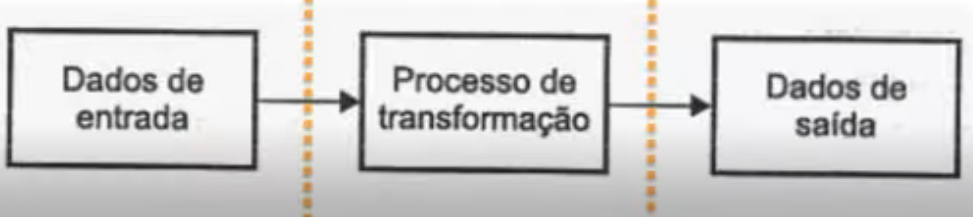
Organização, disposição e ordem dos elementos essenciais que compõem um corpo (concreto ou abstrato).



Não é linear! Posso ter mais de uma opção. Um pouco mais complicada, mas com vantagens.

1. Técnica modular:

Partes independentes que são controladas por um conjunto de regras, sendo que cada módulo tem um conjunto de regras específicas.



Qual a vantagem? A simplificação do algoritmo, da resolução do problema; podemos decompor o problema em problemas menores; e podemos fazer a verificação por módulo.

**Fundamentos de algoritmos:**

**TIPOLOGIA E VARIÁVEIS:**

A função de um computador é processar as informações que passamos para ele. Existem dois tipos de informações: **dados** (objetos de manipulação e de processamento de um computador) e **instruções** (são as normativas e diretrizes utilizadas pelo computador para executar determinadas ações que irão processar aqueles dados)**.**

**Dados** podem ser:

1. Numéricos -> inteiros (todos números inteiros positivos ou negativos) ou reais (todos os números positivos ou negativos com casas decimais);
2. Caractere -> é tudo aquilo que não representamos como número. O próprio número pode ser um caractere;
3. Lógicos -> Verdadeiro (1) ou Falso (0). No Booleano (português estruturado) = verdadeiro (.verdadeiro ou .V ou .S) ou Falso (.falso ou ;.F ou .N).

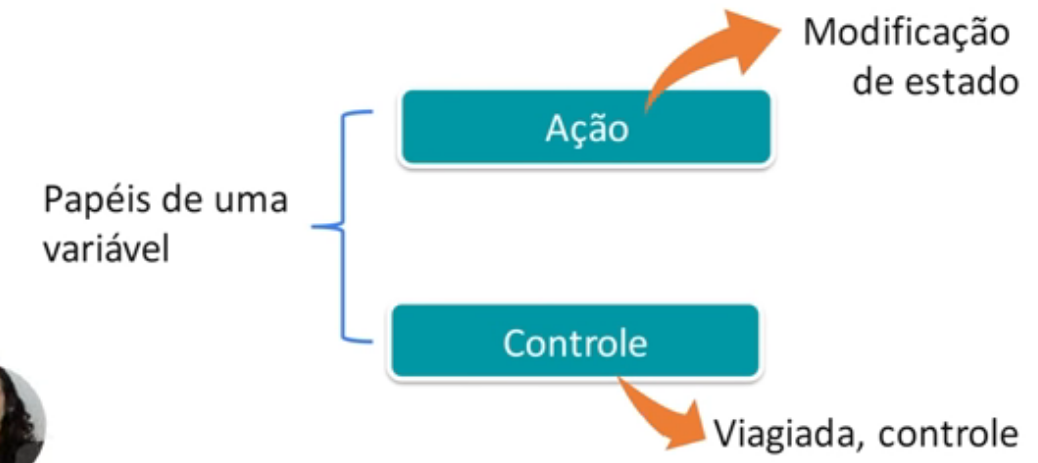
Esses dados são utilizados através de uma **VARIÁVEL**, que é uma estrutura mutável e inconstante, pois pode ser sobescrita, receber mais de um valor e modificar o seu conteúdo.

A variável pode receber qualquer um dos valores de um determinado conjunto de valores. Ex: a – b = d; a + b = c. Contudo, **está restrita ao seu TIPO**. Se é do tipo numérica, vai receber numérica, se é string, irá receber string.

O nome da variável segue algumas boas práticas:

* Atribuição de um ou mais caracteres (não colocar apenas x ou y. Colocar matrícula, CPF, etc. tem que significar algo);
* Primeira letra – não número;
* Sem espaço em branco;
* Vedado a utilização de palavras reservadas para linguagem de programação;
* Pode receber caracteres e números, inclusive booleanos.

Ex de nomes de variável: x2. Nome\_usuario, telefone, user12, Z4.

○

Podemos utilizar uma variável para definir uma constante (que é inalterada), como no caso do pi (π = 3,14), ou resulta = recebido\*0,10.

Os dados são tratados e processados. Já as **INSTRUÇÕES** são **operações** que vão processar esses dados.

**Instruções primitivas:**

As instruções vão determinar as operações que iremos executar em cima de nossos dados. Geralmente, cálculos matemáticos e, para isso, utilizamos **operadores.** Dentro do cálculo matemático, utilizaremos como informação/input são as **variáveis** e as **constantes**.

**Instruções** são linguagem de **palavras-chave** (vocabulário) de uma determinada de programação que tem por finalidade comandar um computador que irá **tratar os dados**.

As instruções primitivas dentro de cada linguagem de programação possuem uma sintaxe particular. E essas instruções também estão ligadas aos operadores.

Expressões executam ações específicas, baseadas em um operador com um ou dois operandos. Um operando pode ser uma constante, uma variável ou um resultado de função. Os operadores são aritméticos, lógicos e relacionais. Como com C, alguns operadores variam em funcionalidade de acordo com o tipo de dados dos operandos especificados na expressão.

**Operadores Aritméticos**

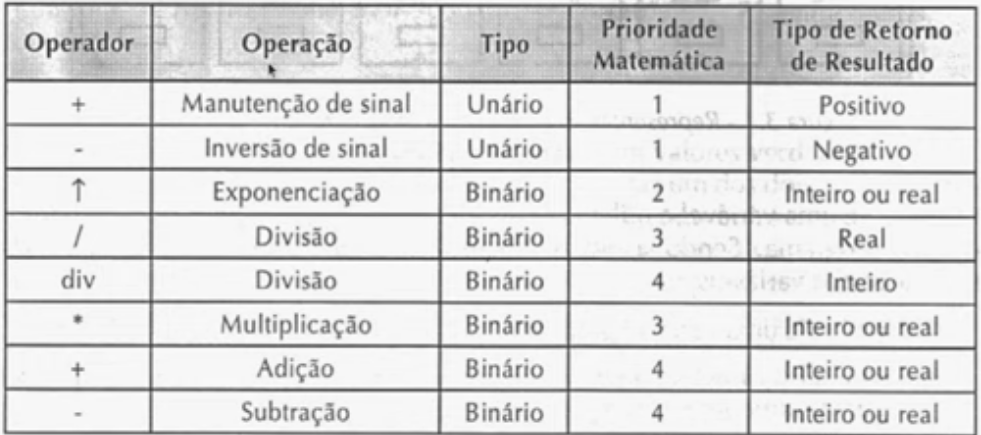
Os operadores aritméticos executam operações matemáticas, como adição e subtração com operandos. Há dois tipos de operadores matemáticos: unário e binário. Os operadores **unários** executam uma ação com um único operando. Operadores **binários** executam ações com dois operandos. Em uma expressão complexa, (dois ou mais operandos), a ordem de avaliação depende de regras de precedência.

Ex: Area = pi\*raio² -> constantes = ² e pi / variáveis = área e raio.

* **Operadores Aritméticos Unários:** Operadores unários são operadores aritméticos que desempenham **uma ação em um único operando**. A linguagem de script reconhece o negativo do operador unário **(-). O operador unário negativo inverte o sinal de uma expressão de positivo para negativo ou vice-versa**. O efeito geral é de multiplicar o número por -1. Por exemplo: a= -10;

**Precedência de Operadores**

As expressões normalmente são avaliadas da esquerda para a direita. Expressões complexas são avaliadas uma por vez. A ordem em que as expressões são avaliadas é determinada pela precedência dos operadores utilizados. A ordenação padrão de C é seguida.



Se uma expressão contiver dois ou mais operadores com a mesma precedência, o operador à esquerda será avaliado primeiro. Por exemplo, 10 / 2 \* 5 será avaliado como (10 / 2) e o resultado será multiplicado por 5.

Quando uma operação de precedência inferior precisar ser processada primeiro, deve ser colocada entre parênteses. Por exemplo, 30 / 2 + 8. Isto é normalmente avaliado como 30 dividido por 2 depois 8 somado ao resultado. Se desejar dividir por 2 + 8, isto deveria ser escrito como 30 / (2 + 8).

Os parênteses podem ser aninhados em expressões. As expressões entre parênteses mais internas são avaliadas primeiro.

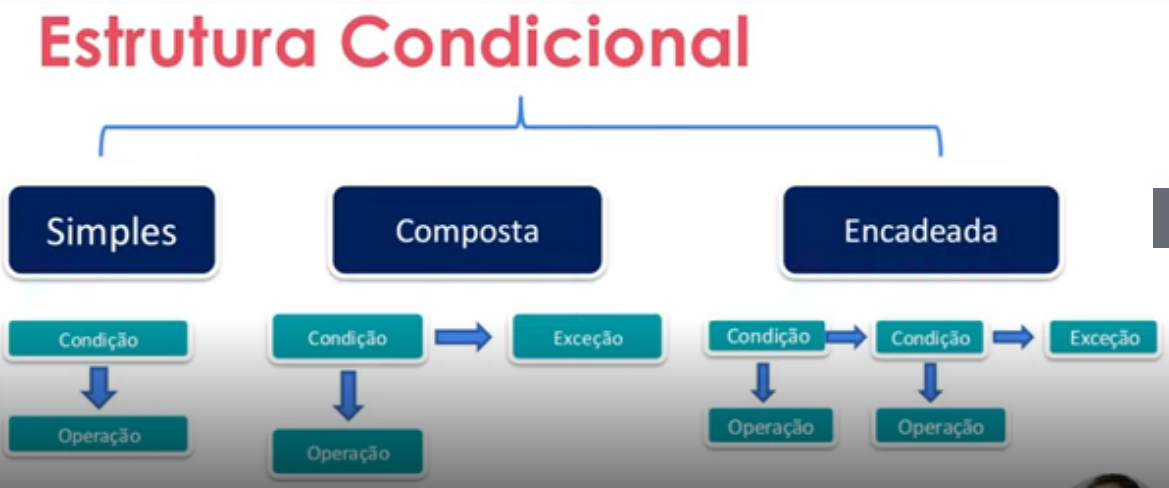
**Estrutura Condicional**

Dado o estado de uma pessoa ou coisa, existe uma condição para aquilo acontecer.

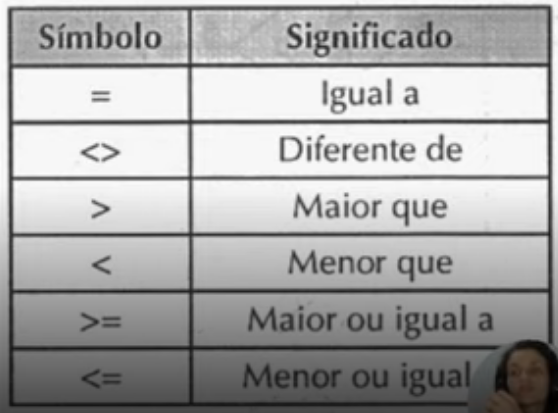
**Condicional expressa uma condição ou suposição/Contem ou implica uma suposição ou hipótese.**

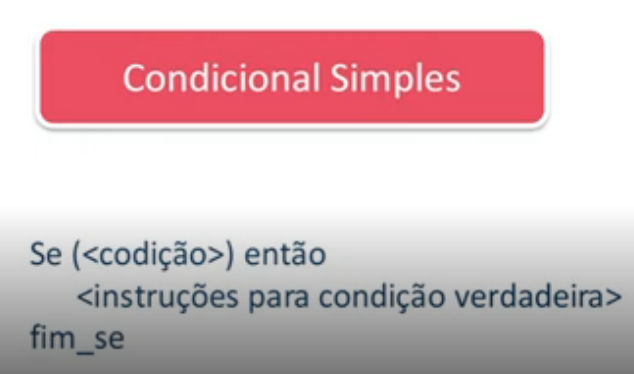
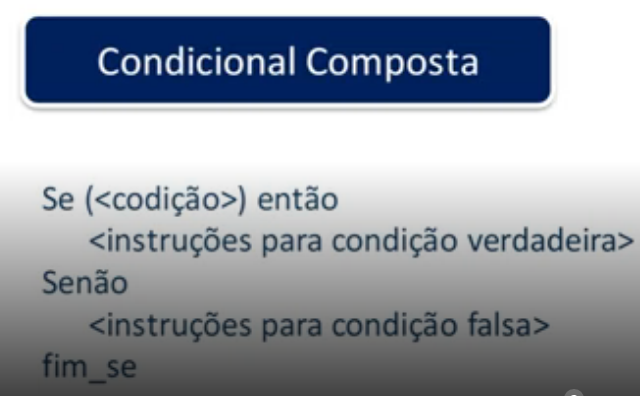


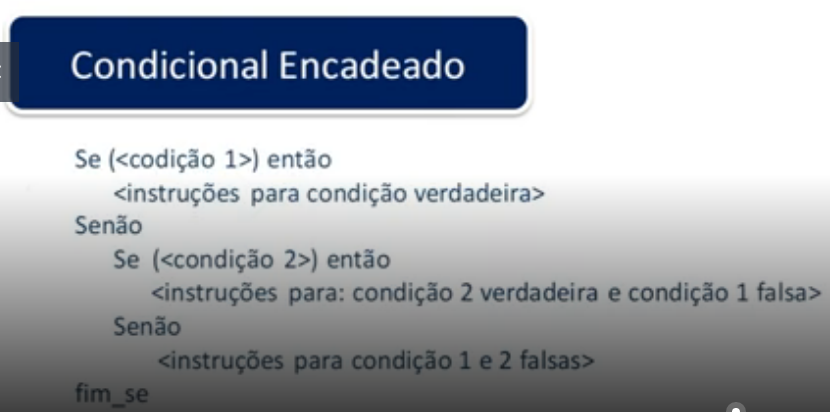
Podem haver diferentes estruturas condicionais:



**Operadores relacionais** que nos ajudam a definir as estruturas de condição dentro de um algoritmo:







**Operadores lógicos** (AND OR NOT)

Quando utilizar? Quando precisa de uma resposta simplificada como VERDADEIRO ou FALSO -> para isso, utilizo o encadeamento de condições.

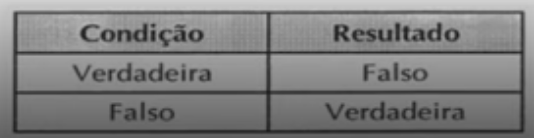
* **AND ->** verifica as entradas, que devem ser satisfeitas. Sempre será **FALSO** se for um ou outro e **VERDADEIRO** quando os dois são satisfeitos. Está associado à **INTERSEÇÃO**.

|  |  |  |
| --- | --- | --- |
| **Condição 1** | **Condição 2** | **Resultado** |
| Falsa | Falsa | Falso |
| Verdadeira | Falsa | Falso |
| Falsa | Verdadeira | Falso |
| Verdadeira | Verdadeira | Verdadeiro |

* **OR ->** apenas **uma das condições** deve ser verdadeira para minha ação ocorrer. Está associado à **UNIÃO**.

|  |  |  |
| --- | --- | --- |
| **Condição 1** | **Condição 2** | **Resultado** |
| Falsa | Falsa | Falso |
| Verdadeira | Falsa | Verdadeiro |
| Falsa | Verdadeira | Verdadeiro |
| Verdadeira | Verdadeira | Verdadeiro |

* **NOT ->** operador de negação. Uma inversão do resultado lógico é o resultado dessa operação.



**Também pode ser usado da seguinte forma: NOT B - > tudo que não está em B.**

**Estruturas de repetição:**

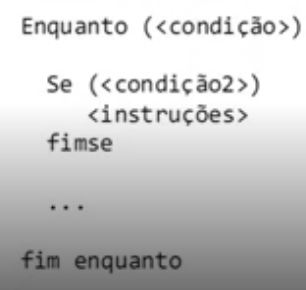
Uma estrutura de repetição irá executar um determinado trecho de programa a partir de certos parâmetros que vão ser estabelecidos dentro dessa estrutura de repetição. Existe, para isso, uma **condição de parada**, para que não entremos em loop:

* Número de repetições pré-fixadas;
* Condição a ser satisfeita.

Vantagens de se utilizar estrutura de repetição: redução de linhas; compreensão facilitada; redução de erro.

OBS: Ver os exemplos nos slides.

Posso mesclar as estruturas (condicional e repetição), utilizando uma dentro da outra. Ex:



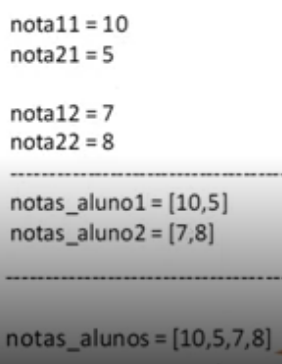
**VETORES E MATRIZES:**

**Vetor** é caracterizado por uma variável dimensionada com tamanho pré-fixado. Ou seja, é uma variável que possui uma sequência, um tamanho pré-fixado, que irá receber n valores. É um container, uma matriz unidimensional.

**Matriz** é uma tabela organizada em linhas e colunas no formato M x N, onde M representa o número de linhas (horizontal) e N o número de colunas (vertical). É uma coleção de variáveis/de vetores. Cada linha pode ser enxergada como um vetor, como uma variável com tamanho pré-fixado, unidimensional.

Para navegar pelas condições de variáveis/vetores, preciso do índice, que determinam a linha e a coluna onde estão cada elemento.

São contíguas em memória, ou seja, armazenadas juntamente na memória do computador.

Ex: 

**FUNÇÕES:**

Função também é chamada de subalgoritmo, método, bloco, subprograma, sub-rotina.

As funções, ou sub-rotinas, são blocos de instruções que realizam tarefas específicas. São trechos de código com tarefas específicas, com objetivos específicos, que podem ser chamados dentro de seu código principal. Vem da ideia de decomposição do algoritmo, de modular o algoritmo e torná-lo mais legível, utilizar a parte de reaproveitamento de código = modularização do problema.

Benefícios: modularização do programa; código mais claro e conciso; reutilização de instruções.

Funções são blocos de instruções (código), identificados por **nomes e parâmetros -> assinatura da função.**

Estrutura: a) definição;

b) Nome;

c) Invocação;

d) variável local -> são destruídas ao encerrar a função.

**INSTRULÇÕES DE ENTRADA E SAÍDA**

ENTRADA = Consiste na inserção e recebimento de dados do mundo real por meio de ação de alguma interface, seja teclado, mouse, arquivos, entre outros.

SAÍDA = consiste na impressão dos dados do mundo abstrato, digital por meio de ação de alguma interface. Os formatos podem variar desde simples arquivos binários até complexas querys de bancos de dados.

Existem dois tipos de saída dentro de um programa (algoritmo):

* Saída programada -> pode ser condicional (aguarda o dispositivo para acionar a saída de impressão do valores) ou incondicional.
* Saída por interrupção -> é definida pelos periféricos.

Os resultados podem ser de casos:

* Bem sucedida;
* Erro de sintaxe ou outro;
* Erros de programação;
* Problemas com a interface.

**INTRODUÇÃO A LINGUAGENS DE PROGRAMAÇÃO:**

Toda evolução da computação partiu do hardware e depois para o software.

O hardware limita a capacidade do software.

Em 1949 surgiu a primeira linguagem de programação -> Assembly. Na realidade, sua classificação correta seria Linguagem de Máquina, pois é uma linguagem de montagem.

Os problemas computacionais são recorrentes. Cada era tem sua dimensão de problemas, mas as raízes dos problemas são similares.

Tipos de problemas: a) Problemas de decisão -> caracter lógico – Sim ou Não. Ideia de pertencimento -> Problema decidível.

b) Problema de busca -> relacionamento binário. O objetivo é semelhante ao nome (“X está em A?”). Recorrente em teoria de grafos (Ex. Clique).

c) Problemas de otimização -> objetivo é maximizar ou minimizar uma função/recursos. Buscam aperfeiçoamento, otimização.

**O que é uma linguagem de programação?**

É um método padronizado composto por um conjunto de regras sintáticas e semânticas de implementação de um **código fonte**.

Temos um conjunto de código fonte que pode ser traduzido ou interpretado.

**Como o computador entende o programa?**

Um programa é um amontoado de palavras se não for possível que o computador entenda.

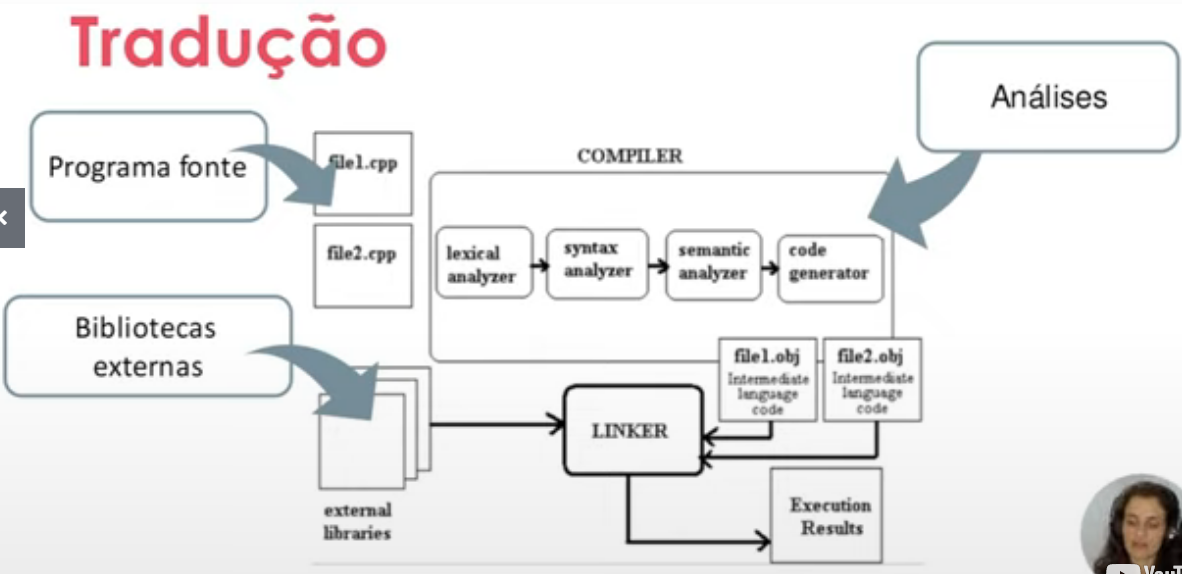
Para isso, existe o processo de tradução do programa, que está relacionado à compilação.

Minha linguagem de alto nível será levada ao compilador, que executará a análise do programa e traduzirá o meu programa de alto nível em um código de baixo nível, de linguagem de máquina, como por exemplo o Assembly.

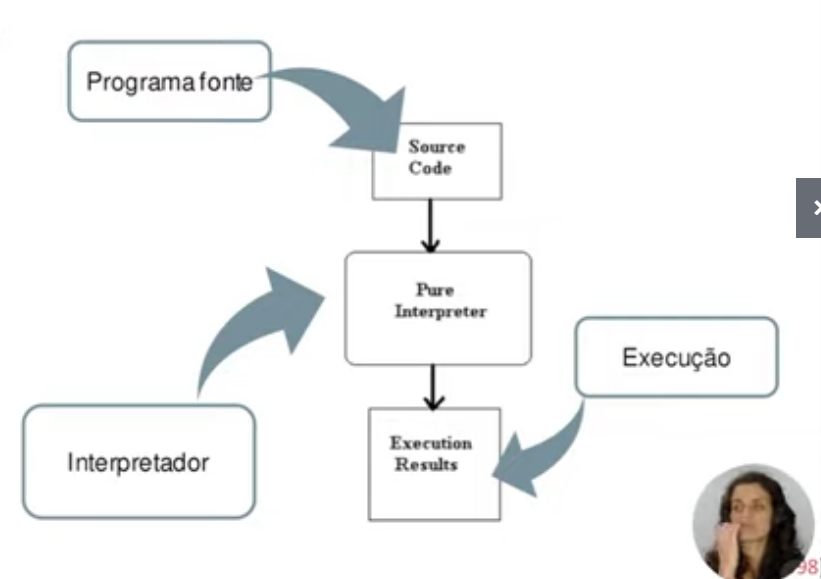
|  |
| --- |
| **Programa de Alto Nível –> Programa Fonte**  **Programa de Baixo Nível –> Programa Objeto** |

Na **TRADUÇÃO**, há a geração do programa objeto e a execução desse programa. É MAIS RÁPIDA e GERA PROGRAMAS MENORES. Ex: Java, C++

Na **INTERPRETAÇÃO,** o programa fonte é executado diretamente, o que torna o processo mais lento. MAIOR FLEXIBILIDADE, MAIS FÁCIL DE PROGRAMAR, MAIS LENTA. Ex: Ruby, Python, JavaScript.



**INTERPRETAÇÃO:**

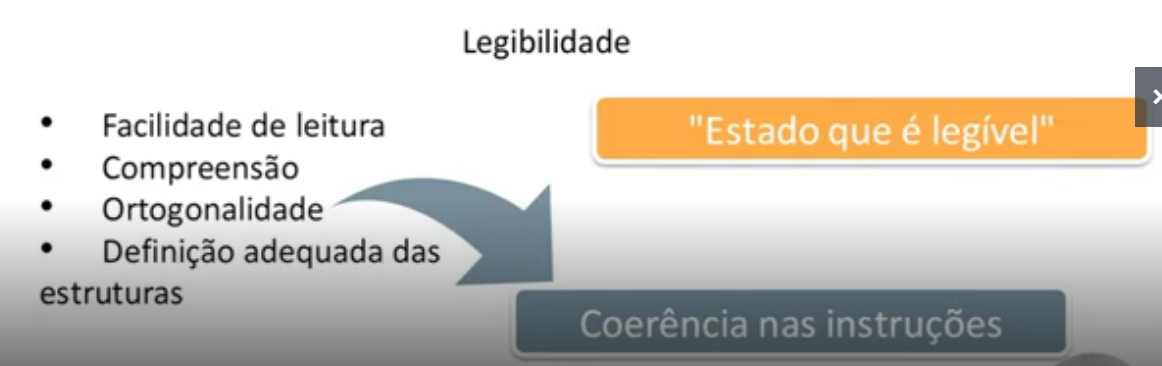


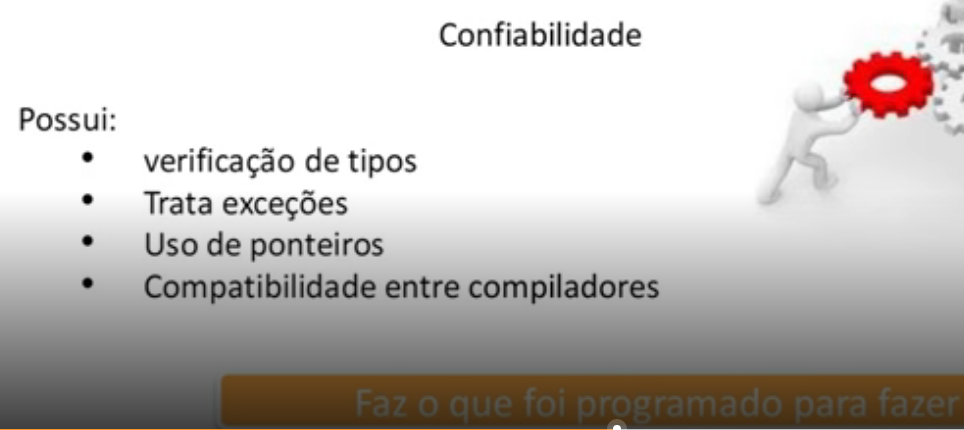
Existe um outro conceito, a **TRANSPILAÇÃO**, que é similar ao conceito de Compilação, só que ao invés de jogar de uma linguagem de alto nível para uma linguagem de baixo nível, eu jogo de uma linguagem de alto nível para uma linguagem também de alto nível, só que um pouco mais baixa. Ex: timescript transpilado para Javascript.

**Características de um programa:**

Diretrizes a serem seguidas para o desenvolvimento de programas/codificação de algoritmos:

* Legibilidade
* Redigibilidade
* Confiabilidade
* Custo





Outras características:

* Atualizações;
* Uso de Inteligência Artificial;
* Disponibilidade de ferramentas;
* Comunidade ativa;
* Adoção pelo mercado.

**Análise de Código:**

1. **ANALISE LÉXICA (Scanner ou Leitura):**

É a primeira fase do processo de compilação, e sua função é fazer a leitura do programa fonte, caracter por caracter, letra por letra, e agrupar os caracteres em lexemas, ou seja, produzir uma sequência de símbolos léxicos, conhecidos como *tokens*.

Nesse processo, irá PARTICIONAR -> **identificar** elementos denominados *tokens* e irá agrupá-los. Esses elementos são: identificadores, palavras reservadas, números, strings, ou seja, todo conteúdo relevante para codificação do programa. E irá **eliminar** caracteres em branco, comentários, tudo o que é usado para entendimento e que o computador não precisa.



1. **ANALISE SINTÁTICA:**

Componente do sistema linguístico que interligam os constituintes da sentença, atribuindo-lhe uma estrutura. É a forma que define, através de palavras reservadas, de indexação, de símbolo especiais, qualquer estrutura relacionada para codificação dentro daquela linguagem específica. Define a corretude do programa.

Ela depende da linguagem de programação utilizada.

1. **ANALISE SEMÂNTICA:**

É p estudo do significado. Incide sobre a relação entre significantes, como palavras, frases, sinais e símbolos. É a lógica do programa.

**Paradigmas de programação:**

PARADIGMA é a forma de resolução de problemas com diretrizes e limitações específicas de cada paradigma utilizando linguagem de programação.

Possui regras para a resolução de um problema e está limitado por diretrizes.

Classificação:

* Orientação a objeto
* Procedural -> chamadas sucessivas e procedimentos separados. Ideia de sequência.
* Funcional -> instruções são baseadas em funções.
* Estruturado -> estrutura de blocos aninhados.
* Computação distribuída -> funções executadas de forma independente. Módulos independentes.
* Lógico

Os mais utilizados são **ORIENTAÇÃO A OBJETO e ESTRUTURADO.**

**PARADIGMA ESTRUTURADO:**

Ideia de sequência, de decisão, de iteração. A ênfase é de instruções sendo executadas em sequência, independente se tem estrutura condicional, de repetição, todas as instruções são executas em sequência.

A decisão está relacionada a um teste lógico. Já a iteração está relacionada a funções, laços, condições.

A Linguagem é C e dominou o mercado até o advento da POO (programação orientada ao objeto).

O C possui uma performance superior, por ser mais baixo nível do que as demais linguagens. Consequentemente, é mais difícil de programar, mas é ótimo para aprendizado e, geralmente, é utilizada em faculdade, mestrado.

Utilização: Problemas simples e diretos; aprender programação.

O POO ainda não é compreendido por muitos.

**ORIENTAÇÃO A OBJETO:**

Paradigma de programação baseado na utilização de objetos e suas interações. Análogo ao mundo real.

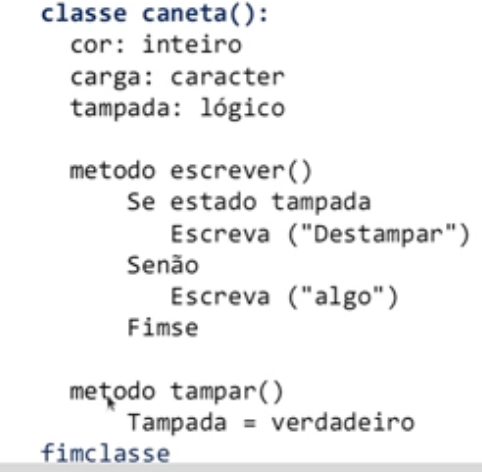
O que é um objeto?

Um objeto é descrito por características específicas (o que eu tenho), comportamentos (o que sou capaz de fazer) e estado (como faço).

CLASSIFICAÇÃO -> MODELO -> ATRIBUTOS/MÉTODOS (operações associadas)/ESTADOS



Observe que na Estrutura, as operações estão desassociadas de uma variável, ao passo que na POO, as variáveis, métodos e estados estão associados ao objeto e, especialmente e somente, àquela classe.



**Pilares de Orientação a Objeto:**

1. Herança -> classe filha herda as características (atributos e métodos) da classe mãe.



1. Encapsulamento
2. Polimorfismo
3. Abstração

Por que utilizar o POO? Para reuso de código. Já para o Paradigma Estruturado (PE) abarca problemas específicos e diretos.

PORTUGOL -> <https://dgadelha.github.io/Portugol-Webstudio/>

**AULA 2 – GIT & GITHUB**

Vantagens do GIT e do GITHUB:

* Controle de versão;
* Armazenamento em nuvem;
* Trabalho em equipe;
* Melhorar seu código;
* Reconhecimento.

**GIT**

É um software de direcionamento de código, que nos ajuda a criar e monitorar diferentes versões do nosso código, dentro de nossa máquina.

Foi criado pelo criador do Linux.

O Git é um sistema distribuído e seguro. É um controle de ver. Versionamento de código distribuído.

**GITHUB**

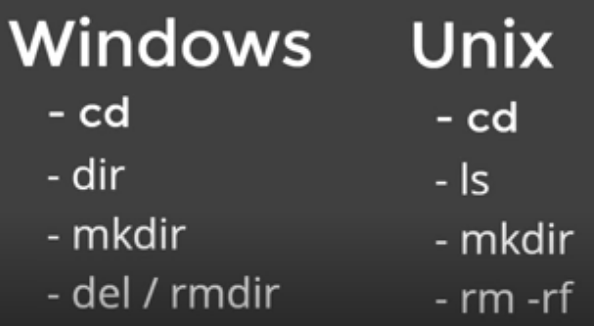
Plataforma em nuvem que guarda os códigos.

**COMANDOS BÁSICOS DE NAVEGAÇÃO NO TERMINAL:**

A maioria dos sistemas operacionais tem programas que possuem uma interface gráfica (**GUI** – Graphical User Interface), ou seja, a forma que o usuário interage com esses programas é de forma gráfica: pode clicar, arrastar, tudo é responsivo aos comandos dos usuários.

Já o software **GIT** tem seu designe voltado para outro tipo de programação, ele é um **CLI** (Command Line Interface), que é diferente do GUI, ou seja, não tem uma interface gráfica. **A forma de interagir com o GIT é por linha de comando.**

**Os comandos usados pelos terminais Windows e Unix (Linux ou Apple) são diferentes:**



Isso tem a ver com a forma que o terminal é. O terminal do Windows é derivado do Shell, e o terminal do Linux é derivado do Bash.

Apertar tecla WINDOWS -> CMD -> PROMPT DE COMANDO (terminal do Windows)

**COMANDOS:**

OBS: TODOS OS COMANDOS ABAIXO POSSUEM FLAGS, ou seja, VARIANTES, que são complementos que acrescentam, modificam ou formatam a forma como esses comandos são devolvidos para a gente.

* **DIR ->** LISTAR. Se situar dentro de um sistema operacional e entender qual local estamos. Traz uma lista de diretórios contidos na pasta no qual estamos situados. Ou seja, dentro de C>users>fe\_fa> tem todas as pastas listadas.
* **CD ->** como desço o nível nesses diretórios? Como vou para uma pasta específica dentro do sistema operacional? Pelo CHANGE DIRECTORY (CD). Comando que possibilita que naveguemos por entre as pastas. CD/ -> leva para a pasta Diretório (C). CD WINDOWS -> para a pasta Windows. Para retroceder, usamos C..
* **CLS ->** Clear Screen. Comando para limpar a tela do diretório.

**ATALHO ÚTIL DO TECLADO -> TAB -> função de autocompletar.** Ex: Se escrever CD W e apertar TAB, ele já reconhece que na pasta que eu estou tem uma pasta chamada Windows, e autocompleta o nome para mim.

* **MKDIR -**> Make Directory. Criar pasta de diretório.
* **ECHO -**> “Printa” de volta no terminal uma frase ou texto. Devo usar o **>** (redirecionador de fluxo), ou seja, pega a saída do meu echo **e joga em um arquivo. Para criar arquivo**: echo hello > hello.txt.
* **ECHO > “nome do arquivo (Ex: README.MD) -**> cria um arquivo
* **DEL -**> deleta ARQUIVOS. Uso para deletar o que tem dentro de uma pasta, mas não deleto a pasta.
* **RMDIR** -> deleta o depositório/a pasta. OBS: escrever rmdir “nome pasta” /S /Q.

OBS: No sistema Linux -> -RM -RF -> R de recursivo (dentro da pasta, pode existir várias outras, o que significa que apagarei todas elas) e o F é de FORCE (para não aparecer nenhum tipo de confirmação para deletar ou não a pasta).

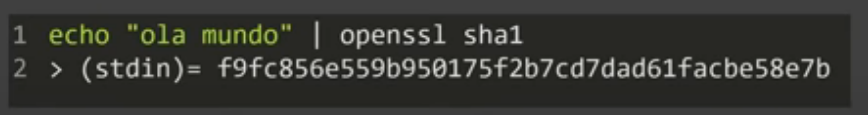
**TÓPICOS FUNDAMENTIAS PARA ENTENDER O FUNCIONAMENTO DO GIT:**

* **SHA1:**

A sigla SHA significa Secure Hash Algorithm (Algoritmo de Hash Seguro), e é um conjunto de funções hash criptográficas projetadas pela NSA (Agência de Segurança Nacional dos EUA). Ou seja, é um algoritmo que irá pegar um arquivo e vai embaralhar de uma forma muito específica.

Isso é relevante para nós, pois a encriptação (saída dos dados) gera um conjunto de caracteres identificador único de 40 dígitos. Cada modificação que eu faço no meu arquivo, gera um novo código, o que me permite identificar cada arquivo gerado de uma forma segura e rápida.

É uma forma curta de representar um arquivo.



Usar o GIT BASH.

Usar o comando OPENSSL SHA1 “nome do arquivo”

Toda vez que eu mudo o arquivo, ele gera um novo número de códigos. Porém, se eu apago a mudança e salvo, o código gerado será o mesmo anterior àquela mudança feita. Ex:

1° arquivo feito -> código 1

2 – Acrescentei uma virgula no arquivo -> código 2

3 – apaguei a vírgula, ficando igual ao primeiro arquivo -> código 1.

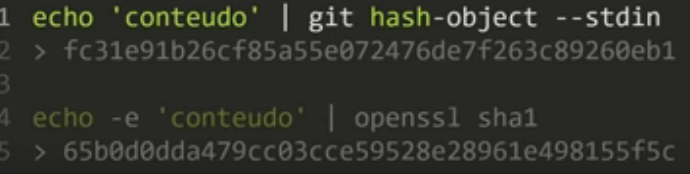
O sha1 também é gerado para os objetos internos do GIT.

**Objetos Internos do Git:**

1. **Blobs ->** “bolha”**.** Um blob (do inglês: Binary Large OBject, basic large object, BLOB ou BLOb, que significa objeto grande binário ou objeto grande básico na tradução literal), é uma **coleção de dados binários armazenados como uma única entidade em um sistema de gerenciamento de banco de dados**. Blobs geralmente são objetos de imagem, áudio ou outros objetos multimídia, **apesar de algumas vezes código binário executável ser armazenado como um blob**. O suporte de blobs por bancos de dados não é universal**.**

BLOBs têm diversas utilidades mas o uso mais frequente é a armazenagem de dados em bancos de dados. Normalmente os bancos de dados provêm suporte para diversos tipos básicos de informação como strings, números, datas, e outros, para que o banco possa conter dados que não estão cobertos neste suporte estes dados são armazenados como um BLOB.

São responsáveis pelo versionamento do nosso código.

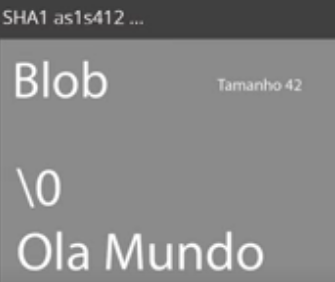


Função **echo** vai pegar uma string e “cuspir” o output dessa string, e vamos passar o output, que no caso é “conteúdo”, para uma função do GIT. Para isso, usamos a função **hash-object** e a flag **--stdin** demonstra que essa função espera receber um arquivo, e a gente está enviando um texto.

Sendo assim, quando a gente passa a string “conteúdo” para essa função do GIT, ela devolve a sha1 desse “conteúdo”.

Se eu usar o comando “openssl” especificando o sha1, será gerada uma outra chave sha1. Isso acontece porque o GIT lida e manipula os arquivos como objetos, que é chamado de **“blob”**. O blob contém **metadados.**

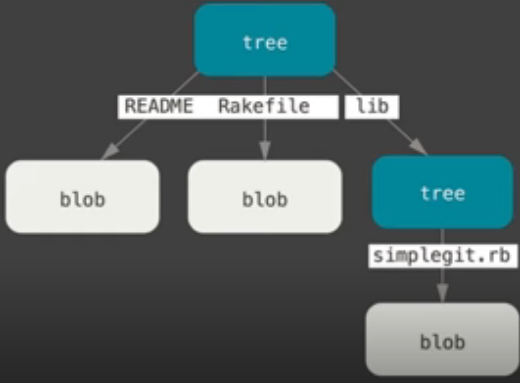
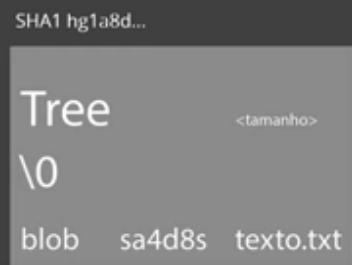
**O objeto blob conterá: o tipo do objeto (blob), um tamanho desse arquivo ou string, \0 e o conteúdo de fato do arquivo (seja texto, binário, etc):**



Quando eu detalho as especificações do blob ‘conteúdo’, eu consigo encontrar a mesma chave gerada em git em sha1:



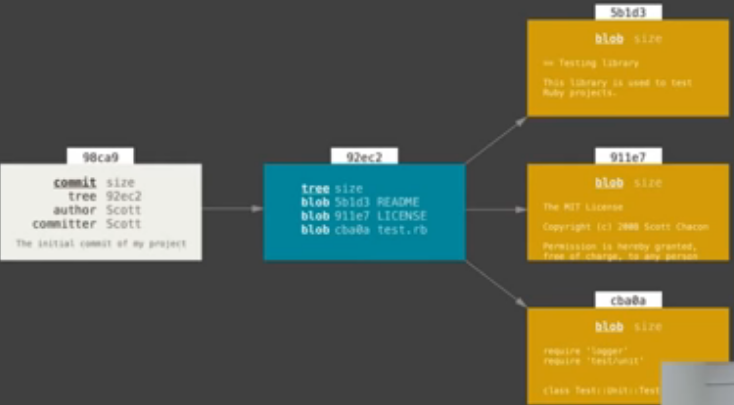
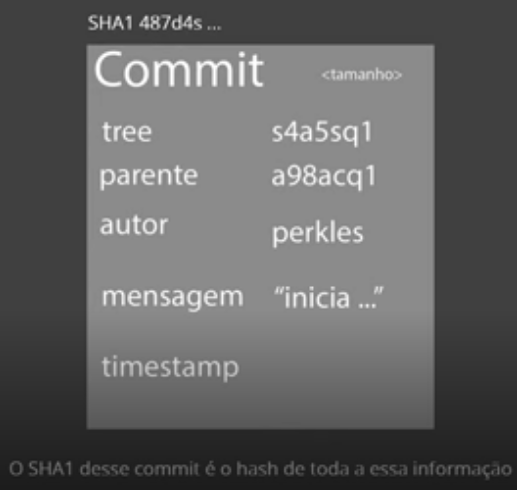
1. **Trees ->** as “trees” (árvores) armazenam “blobs”. A blob é o bloco básico de composição e a tree armazena e aponta os tipos de blob diferentes do arquivo. A tree também contém metadados (\0), aponta para um blob, que por sua vez tem um sha1, e a árvore também guarda o nome desse arquivo, ao contrário do blob, que só guarda o sha1 do arquivo (que são os caracteres identificadores). A árvore é responsável por montar toda a estrutura e onde está localizado os arquivos. Elas podem apontar para blobs (arquivos) ou outras árvores. Isso é possível porque os diretórios de um sistema operacional podem conter outros diretórios. Por isso, blobs são objetos recursivos. **As árvores também possuem um sha1 desse metadado**. Portanto, se eu mudar uma vírgula no arquivo blob, além de mudar o sha1 desse blob, também irei mudar o sha1 do metadado no qual ele se encontra, que existe dentro da árvore.



1. **Commits -> é o objeto mais importante, pois irá juntar tudo e dará sentido às alterações que se está fazendo.** O Commit aponta para uma árvore, aponta para um “parente”, ou seja, para o último “commit” realizado antes dele. Aponta para um autor, para uma mensagem. O autor e a mensagem fazem parte dessa ideia de “sentido” já falada. As mensagens podem ser escritas no objeto commit para explicar, dar significado para esse monte de arquivo contido nessa grande pasta. O autor é quem está fazendo o commit. Por sua vez, o timestamp (carimbo de tempo) indica a hora na qual o commit foi criado.

Os commits também possuem um sha1, que agem da mesma forma que nas trees e blobs.

Quando se tem um commit, você garante que ninguém alterou aquele commit. É uma forma segura de demonstrar que aquele commit representa exatamente o que você quer dizer com o código, que não teve interferência de outras pessoas, pois não tem como alterar o código de um commit ou de qualquer arquivo dentro dele sem que seja demonstrado no histórico de commits.

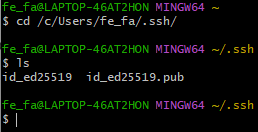


**CHAVES SSH e TOKENS:**

* **CHAVE SSH** é uma forma de estabelecer uma conexão segura e encriptada entre duas máquinas. No caso, o servidor do GitHub e a máquina local. **Sempre será 2 chaves: uma pública e uma privada**. A chave pública será colocada no GitHub. Todos os depositórios que contiver na nossa máquina por esse processo ssh e for enviado no GitHub, ele já irá conhecer a assinatura da máquina e, portanto, serei capaz de enviar o código sem precisar de senha.

Gerar chave SSH para Windows e Linux é bem parecido, por conta do GitBash.

Os comandos no GitBash para gerar as chaves são: ssh-keygen -t ed25519 -C [fernandaathanazio@gmail.com](mailto:fernandaathanazio@gmail.com) (dar enter). Depois é só colocar uma senha e dar enter. A chave ficará na pasta /c/Users/fe\_fa/.ssh/id\_ed25519 (o . significa que a pasta é oculta). Depois, no próprio Git, limpar diretório (Ctrl+L) e escrever:



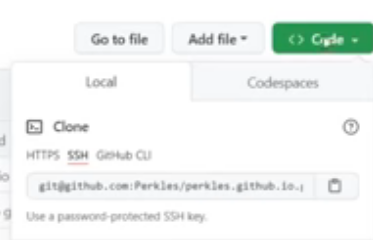
* Comando para visualizar conteúdo de umas das chaves: cat id\_ed25519.pub

A chave que aparecer é a que será colocada no GitHub

Agora preciso inicializar o **ssh-agent**, que é uma entidade que vai pegar as chaves e lidar com elas. Comando usado é: eval “$(ssh-agent -S)”. Isso irá gerar um **agent pid** (agente pid 1181).

Agora devo escrever: ssh-add id\_ed25519. Vou usar a chave PRIVADA, pois toda vez que chegar uma criptografia com essa chave, o programa irá usar a chave privada para descriptografar essa mensagem. O agente ficará encarregado de perceber esse tipo de coisa.

OBS: Quando eu clono um arquivo no BitHub, devo copiar o endereço SSH:



Essa chave eu colo no GIT com o comando git clone.

* **TOKEN DE ACESSO PESSOAL** tem um início bem parecido (terá que ser gerado no GitHub), mas se assemelha mais ao processo de digitar nickname e senha do que a vantagem de uma chave ssh, de não requerer senha.

Gera-se um token no GitHub e guarda na sua máquina ou em algum lugar e, sempre que for fazer um commit, o Git vai pedir seu usuário e senha, aí coloca o usuário do GitHub e na senha coloca o token de acesso pessoal.

Vai em GitHub -> Configurações -> Developer Settings -> Personal acess tokens -> Fine-grained tokens -> Generate new token.

Copia o token que irá gerar e salva em algum lugar do computador.

Nesse caso, quando for clonar um arquivo do GitHub, usa-se o endereço HTTPS.

**COMANDOS DO GIT:**

Todo comando GIT precisa levar o seu nome na frente, para chamar o comando ao terminal. Depois vem o comando específico.

- **git init** -> iniciar o repositório GIT;

- **git add** **“nome do arquivo”** -> mover arquivo e dar inicio ao versionamento e conhecer os primeiros comandos;

- **git add \* ou git add . ->** adicionar tudo o que está no working directory e adiciona para staged, para podermos commitar.

- **git commit** **-m “frase que explica a alteração para novo commit”** (Ex: adiciona index) -> criar o primeiro commit;

- **git commit -m “commit inicial”** -> criar um commit inicial

- **ls -a** -> mostra arquivos ocultos;

**- cd /\*nome da pasta que eu quero\*** -> muda a pasta

**- cd ..** -> volta pra pasta superior

- **git config --global user.email “**[**fernandaathanazio@gmail.com**](mailto:fernandaathanazio@gmail.com)**”** e **git config --global user.name Fernanda** -> configurar git antes de começar a usar

- **Adicionar arquivo** -> vamos adicionar um arquivo MARKDOWN (que é uma forma mais humana de se escrever um artigo HTML). Sempre terá a extensão MD no final (Ex: strogonoff.md).

OBS: Para saber todas as funções e marcações de formatação do texto markdown, ir em Typora -> Ajuda -> Markdown References

* **GIT INIT** -> estamos criando um repositório GIT dentro do depositório/pasta selecionada. Inicializa o conceito de REPOSITÓRIO no Git.
* **TRACKED E UNTRACKED ->** Arquivos que podem ser rastreáveis são os Tracked. Podem se dividir em Unmodified, Modified e Staged. Já os Untracked são os arquivos que ainda não temos ciência.

1. Unmodified -> arquivos que ainda não foram modificados;
2. Modified -> arquivos que sofreram modificação;
3. Staged -> são os arquivos que estão se preparando para fazer parte de outro agrupamento.



*Quando criamos o arquivo “Strogonoff”, ele estava* ***UNTRACKED****, o Git ainda não sabia de sua existência. Quando rodamos o comando “****git add****”, o Git moveu o arquivo untracked direto para* ***STAGED****.*

*O Git sabe que um arquivo passou de Unmodified para Modified quando compara os sha1 de ambos.*

*Se usarmos o “git add” no arquivo Modified, ele também vai direto para a Staged.*

*E quando removemos um arquivo Unmodified, sem sofrer nenhuma alteração, ele volta para a categoria Untracked.*

*Quando movemos um arquivo Modified para Staged, ele está se preparando para fazer parte de um* ***COMMIT****.*

*O Commit retorna todos esses arquivos para Unmodified, para começar o ciclo de novo. O Commit salva e cria um* ***SNAPSHOT*** *(salvar/tirar uma foto) do código naquele momento.*

**Quando rodo o GIT INIT, eu inicio um repositório.**



O **AMBIENTE DE DESENVOLVIMENTO** representa tudo que está na nossa máquina, e o **SERVIDOR** é a versão distribuída. O Git é um SISTEMA DISTRIBUÍDO, ou seja, terá uma versão no servidor (GitHub), e tem a versão da máquina. Sendo assim, as alterações do código feitas na máquina não repercutem imediatamente na versão que está no **repositório remoto (Remote Repository)**. Somente quando se executa um conjunto de códigos específicos, se empurra a alteração do repositório local ao repositório remoto.

Repositório de trabalho – Working Directory

Local onde os arquivos staged ficam - Staging Area

Assim, os arquivos ficarão sempre alterando entre o repositório de trabalho e o “staged area” a medida que se vai adicionando novos arquivos nesse trabalho e modificando novos arquivos nesse repositório.

**Quando se faz um commit**, o arquivo passa a integrar o **REPOSITÓRIO LOCAL (Local Repository)**. E o repositório local, por sua vez, pode ser mandado para um Repositório Remoto. O repositório local só será e deverá ser composto por commits.

* **GIT STATUS ->** demonstra os status do arquivo (unmodified, modified e staged). Comando usado com muita frequência.
* **MV “nome do arquivo/pasta” ./”nome do outro repositório” (Ex: mv strogonoff.md ./receitas/) ->** mover arquivo ou pasta para outro local.

**Obs: toda vez que modificar o arquivo ou coloca-lo em outro local, usar: git add para manda-lo para staged.**

* **GIT RM “nome do arquivo” ->** mover arquivos para unmodified.
* **GIT RESTORE --STAGED “NOME DO ARQUIVO” ->** retirar os arquivos do modo staged.
* **GIT CONFIG --LIST ->** lista todas as configurações específicas do meu git.
* **GIT CONFIG --GLOBAL --UNSET USER.NAME/USER.EMAIL ->** para deletar o nome de usuário e email. Após, devo configurar novo usuário e/ou email.

Quando tudo estiver pronto, ir no GITHUB e criar um arquivo. Quando o arquivo for criado, ele gerará um endereço de HTTP (Ex: <https://github.com/FernandaAthanazio/livro-receitas.git>). Após, ir no GIT e adicionar nosso repositório local para um repositório remoto. Para isso, temos que adicionar primeiro a origem na qual estamos enviando esses arquivos no Git (endereço criado pelo GitHub). Usamos o comando **GIT REMOTE ADD ORIGIN “ENDEREÇO HTTPS”.**

* **GIT REMOTE ADD ORIGIN “ENDEREÇO HTTPS GERADO GITHUB” ->** cadastra um repositório remoto.

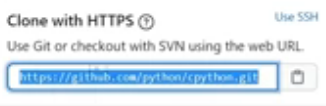


* **GIT REMOTE -V** -> lista as listas de repositórios remotos cadastrados.
* **GIT PUSH ORIGIN MAIN** -> envia o diretório local para o remoto.

**Como resolver conflitos:**

* **Conflito de Merge** -> quando o GitHub tenta juntar os códigos de duas alterações, mas existem alterações na mesma linha. Nesse cenário, o GitHub não irá fazer nenhuma alteração drástica, ele espera que a pessoa abra o arquivo que teve mais de uma edição na mesma linha por duas ou mais pessoas diferentes, e a pessoal, manualmente, resolva o conflito e depois empurre o código para o GitHub.
* **GIT PULL ORIGIN MAIN** -> puxa as duas ou mais versões do arquivo para um único. Se der algum conflito de merge, entrar no arquivo, modificar e salvar a versão final. Após, git add \* e git commit -m “resolve conflitos”. Após empurra pro GitHub (git push origin master).

Para copiar um arquivo de terceiro e você modifica-lo:

1. Pegar o endereço HTTPS do arquivo: ;
2. Ir no GitBash;
3. Abrir o diretório que deseja colocar o arquivo (ex: pasta workspace);
4. Usar código: **git clone *endereço.***