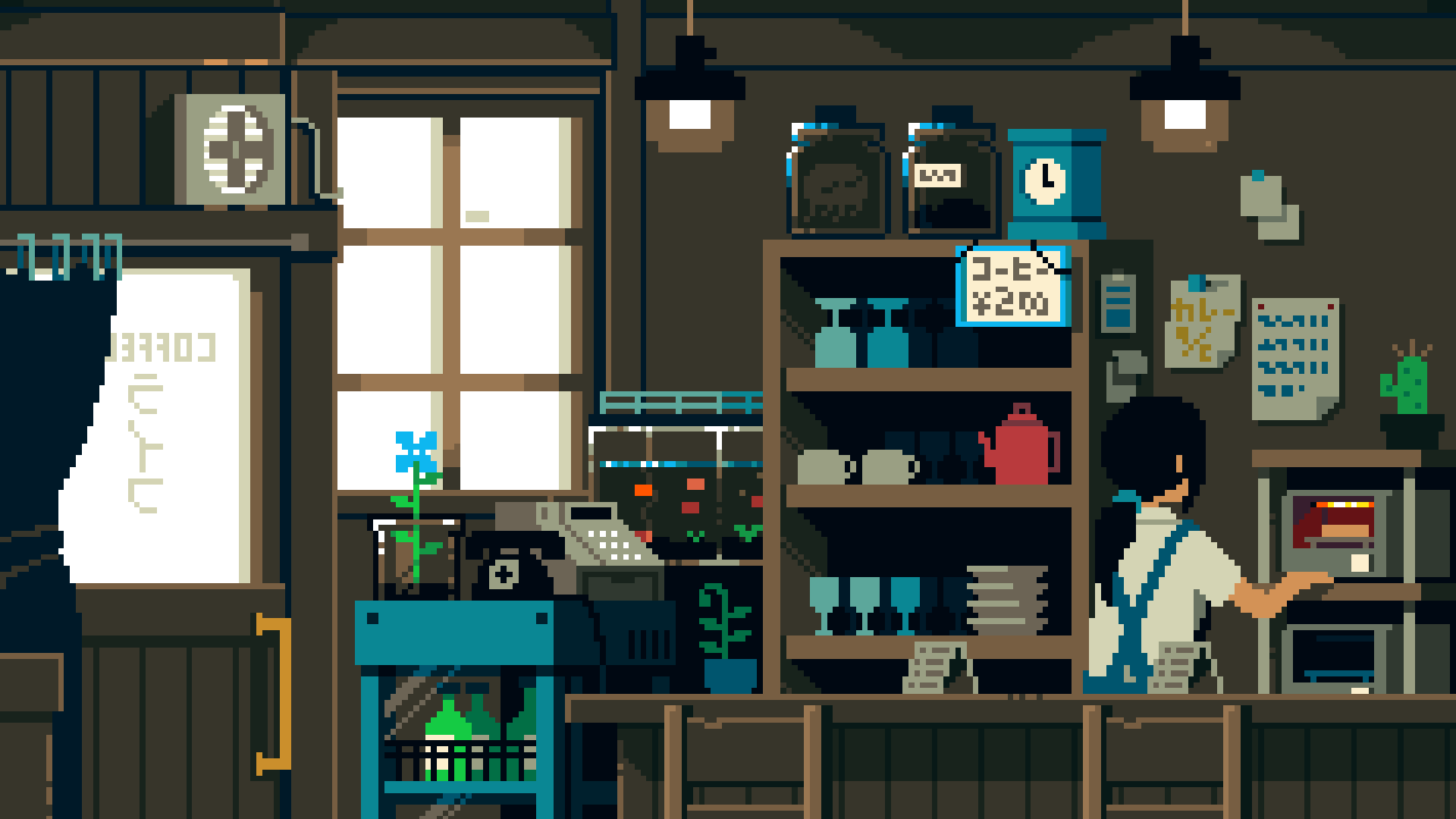
Digital Innovation One – bootcamp Decola Tech



**TREINAMENTO 1: LÓGICA DE PROGRAMAÇÃO ESSENCIAL**

**Bloco 1: Introdução à lógica de programação**

**Entendendo a lógica**

- Lembrar que programar é resolver problemas!!

- “Abstração” é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes

**Algoritmos e pseudocódigo**

- Algoritmo é uma sequência de passos que resolve um problema

Início-dia

Acordei

Levantei da cama

Troquei de roupa

Escovei os dentes

Fui à padaria

Tomei café

...

Deitei na cama

Dormi

Fim-dia

- O “pseudocódigo” é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples

**Fluxogramas e variáveis**

- Fluxograma é uma ferramenta utilizada para representar graficamente o algoritmo, isto é, a sequência lógica e coerente de dados

- Na programação, uma **variável** é um objeto (uma posição, frequentemente localizada na memória), capaz de reter e representar um valor ou expressão

- As **constantes** são valores imutáveis e não são alterados ao longo da vida do programa

**Tomadas de decisões e expressões**

- As **expressões literais** são expressões com constantes e/ou variáveis que tem como resultado valores literais; as usamos na atribuição de valor para uma variável ou constante

- As **expressões relacionais** são compostas por outras expressões ou variáveis numéricas com operadores relacionais (retornam valores lógicos verdadeiro/falso)

- Grande parte da programação é feita a partir das **tomadas de decisão**.... Geralmente, quando escrevemos um programa acontece a necessidade de decidir o que fazer dependendo de alguma condição encontrada durante a execução

**Concatenação**

- Este é um termo utilizado em computação para designar a operação de unir o conteúdo de duas strings

OU

- Agrupamento de duas ou mais células que, incluindo fórmulas, textos ou outras informações contidas no seu interior, dá origem a um **único resultado**

**Bloco 2: Introdução ao Portugol**

**Estrutura de repetição**

- O que é? É uma estrutura que permite **executar mais de uma vez** o mesmo comando ou conjunto de comandos, de acordo com uma condição ou um contador

**Linguagens de programação e Portugol**

- Linguagem de Programação é uma linguagem escrita e formal que especifica um conjunto de instruções e regras usadas para gerar programas (software)

- Um software pode ser desenvolvido para rodar em um computador, dispositivo móvel ou em qualquer equipamento que permita sua execução

- As linguagens servem como um meio de comunicação entre computadores e seres humanos

**Alto nível x Baixo nível**

- As linguagens de **alto nível** são aquelas que a sintaxe se aproxima mais da nossa linguagem e se distancia mais da linguagem de máquina, ex.: Python, C#, C++...

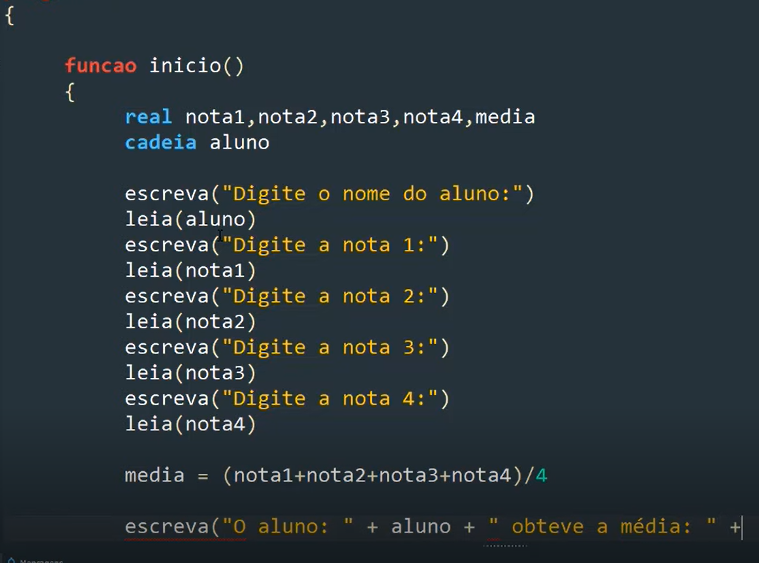
- Já as de **baixo nível** são as que a sintaxe se aproxima mais da linguagem de máquina (necessário ter o conhecimento direto da arquitetura do computador para fazer alguma coisa), ex.: Asembly

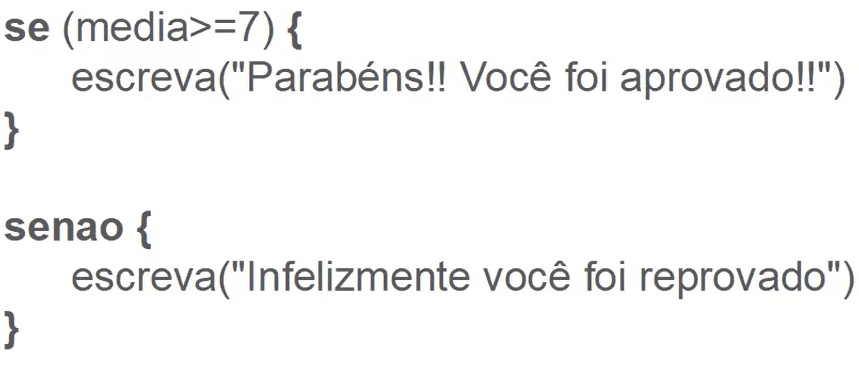
**Compiladas x Interpretadas**

- As **compiladas** são as linguagens em que o código fonte é executado diretamente pelo sistema operacional ou pelo processador, após ser traduzido por meio de um processo chamado compilação, ex.: C#...

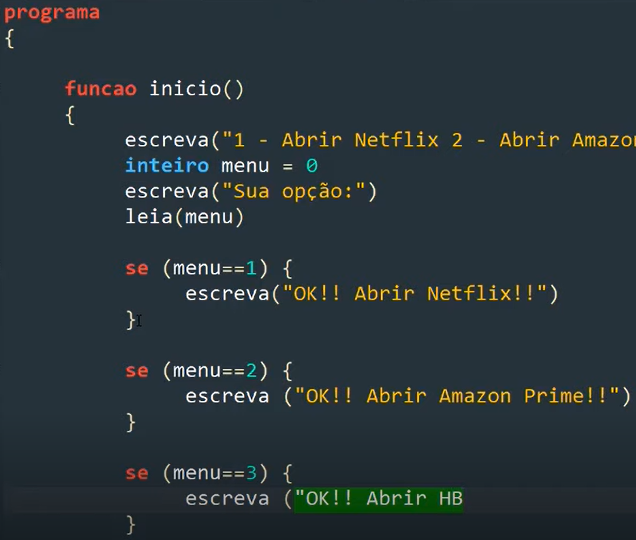
- As **interpretadas** são linguagens em que o código fonte é executado por um programa de computador chamado interpretador que em seguida é executado pelo sistema operacional, ex.: JavaScript, PHP, R, Python...

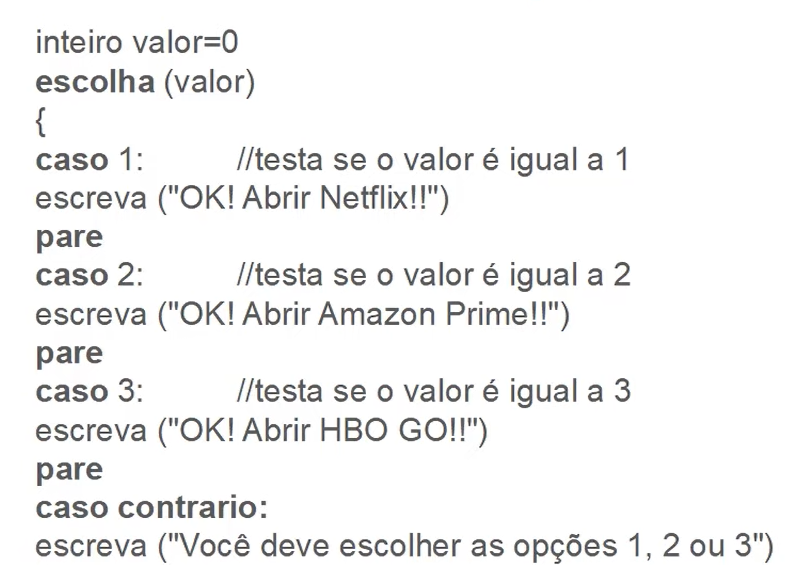
**Portugol...**





- No Portugol, comentário se inicia com //





**TREINAMENTO 2: ESTRUTURA DE DADOS E ALGORITMOS**

- O que é uma **estrutura de dados**?

É uma estrutura organizada de dados na memória do pc ou em qualquer dispositivo de armazenamento, de forma que os dados possam ser utilizados de forma correta

- As estruturas encontram muitas aplicações no desenvolvimento de sistemas, sendo que algumas muito especializadas e utilizadas em tarefas específicas (como mecanismo de busca e análise de bancos de dados)

- **Algoritmo** é um conjunto de instruções estruturadas e ordenadas, seu objetivo é realizar uma tarefa ou operação específica

- As **principais estruturas de dados** utilizadas em desenv. de sistemas são:

Vetores e matrizes (arrays)

Registro

Lista

Pilha

Fila

Árvore

Tabela Hash

Grafos

**Arrays**

- São estruturas de dados simples que podem auxiliar quando há muitas variáveis do mesmo tipo em um algoritmo

- O **vetor (array uni-dimensional)** é uma variável que armazena várias variáveis do mesmo tipo; é uma estutura indexada que pode armazenar uma determinada quant. de valores

- \t dá espaço, enquanto \n pula linha

- A **matriz (array multidimensional)** é um vetor de vetores, possuindo duas ou mais dimensões

**Registros**

- É uma estrutura que oferece um formato especializado para o armazenamento de dados

- Diferentemente dos arrays, o registro permite armazenar mais de um tipo de dado

- Toda estrutura de registro tem um nome (ex: livro) e seus campos podem ser acessados por meio do uso do operador ponto, ex.:

Para acessar o preço de um livro, poderíamos utilizar a declaração **livro.preco**

**Listas**

- Armazenam dados de um determinado tipo e numa ordem específica

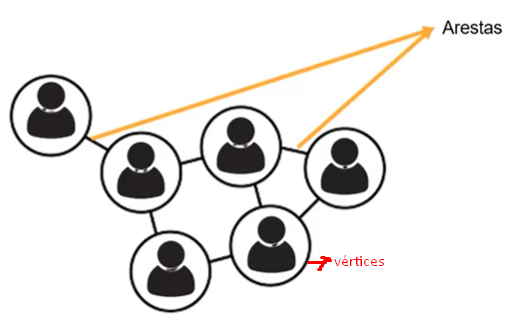
- Mas qual a diferença para os arrays? É que as listas possuem ajustável, enquanto os arrays possuem um tamanho fixo

- Existem dois tipos de listas: ligadas e duplamente ligadas

- Aqui não anotei PILHA, ÁRVORES E TABELAS HASH (tabela hash me lembra dicionários em pytho, porque associa uma chave a um valor)

Grafos

- Estruturas que permitem programar a relação entre objetos (objetos são vértices ou nós do grafo)



**TREINAMENTO 3: INTRO AO GIT E GITHUB**

- Controle de versão

- Armazenamento em nuvem

- Melhoria do código

- Reconhecimento

**Comandos básicos para um bom desempenho no terminal**

- A maioria dos programas possuem interface gráfica, mas o GIT é um CLI e a forma de interagir com ele é por linha de comando



- No Windows, para limpar o terminal usamos o comando **cls** (clear screen)

“Silêncio é sucesso!”

**Entendendo como o GIT funciona**

- O SHA1 é um conjunto de funções hash criptografadas projetadas pela NSA -> essa encriptação gera conjunto de caracteres IDENTIFICADOR de 40 dígitos

- O GIT usa esse identificador sha1 para identificar através dessa sequência de caracteres cada versão de um arquivo que a gente salva

**Objetos internos do Git**

**Blobs**

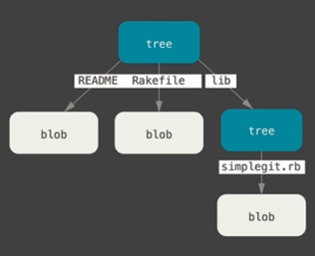
- O git organiza os arquivos dentro de um objeto do tipo blob que contém metadados (como o tamanho do arquivo)

-

**Trees**

- Também contém metadados e aponta para um blob (este, por sua vez, tem um sha1)

- As árvores, por sua vez, têm um sha1 para os metadados desta árvore! Ou seja, se mudar uma vírgula do arquivo que está ali, muda também o sha1 da árvore



**Commits**



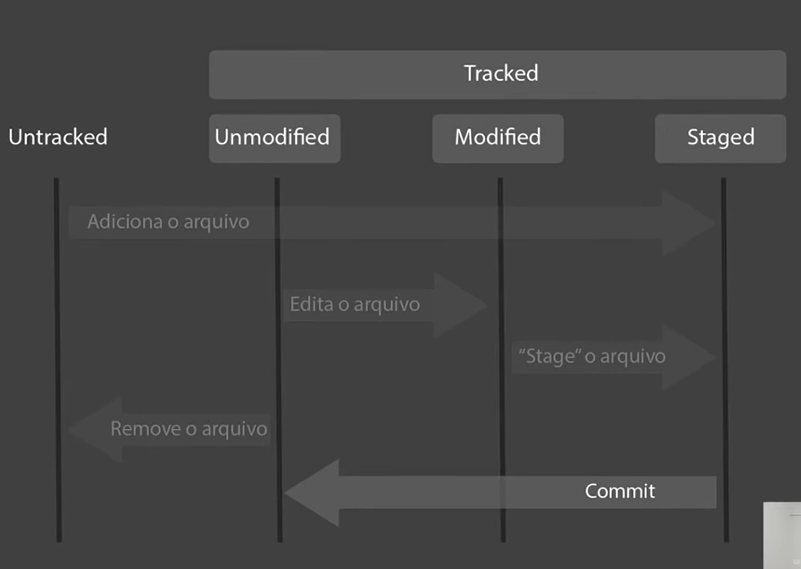
- Os commits também possuem um sha1

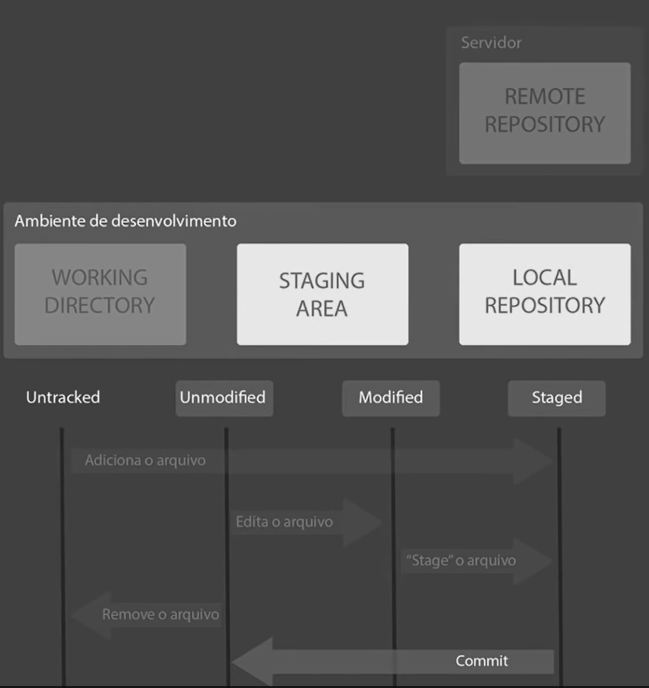
- Eles devem receber um nome para explicar porque os sha’s ali dentro estão “alterados”

**Chaves SSH e Token**

Jesus??????????

**Ciclo de vida**





- Para clonar um repositório direto do Git Bash usamos o comando:

git clone <url do repositório>

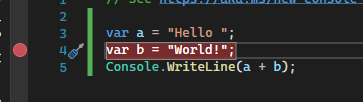
**TREINAMENTO 4: IDE Instalação e Configuração**

**Gerando o código e Debugging**

- A depuração é feita quando é colocado um breakpoint no código e ele é executado

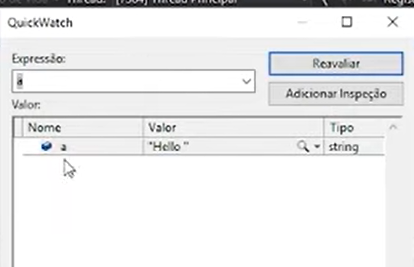
- É possível navegar pelo código no depurador usando comandos!

- Aperta F9 numa linha para o código parar de rodar naquela linha; depois F10 para rodar a próxima linha ou F5 para rodar todo o resto do código;



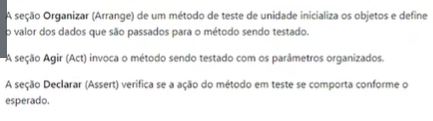
- A inspeção deve ser feita na depuração

- Pode ser feita passando o cursor do mouse por cima da variável ou abrindo a janela QuickWatch pelo menu Depurar



**Testes**

- Para criar um teste, devemos clicar no método que vai ser testado com o botão direito e então em criar testes de unidade



**TREINAMENTO 5: Introdução ao C# e .NET**

**O que é .NET?**

- É um framework criado pela Microsoft que permite a comunicação entre várias linguagens que tem uma infraestrutura comum (CLI), entre elas o Visual Basic, C++, F# e C#

**O que é C#?**

- É uma linguagem orientada a objetos **fortemente tipada**. A sintaxe orientada a objetos foi baseada no C++ e possui influências de outras linguagens de programação como Object Pascal e Java

**O que é uma solução?**

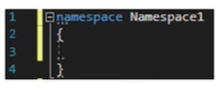
- Por definição, a solução é um *container* para organizar os projetos relacionados

**Estruturas**

**Namespaces**

- Os namespaces são usados para organizar classes

- Para usar uma classe de outro namespace, colocamos using no começo do código



**Classes**

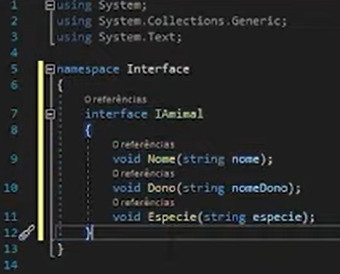
- Uma classe pode ter campos, propriedades, métodos e eventos dentro dela que são denominados **membros**

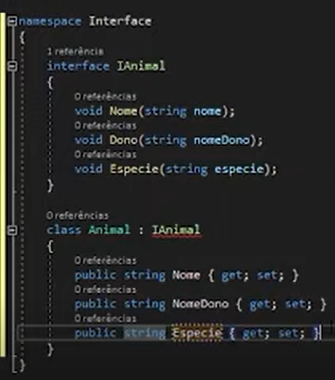
- Quando uma classe é **instanciada**, essa instância é chamada de OBJETO e nesse objeto podem ser criados os **valores** de acordo com os campos da classe

**Interface**

- É como uma classe base que serve de molde para várias classes

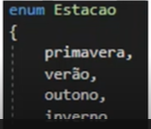
- Obrigatoriamente, uma classe que implementa uma interface deve implementar todos os membros da classe





**Enum**

- Declara um conjunto de constantes nomeadas que começam do 0 e aumentam de 1 em 1



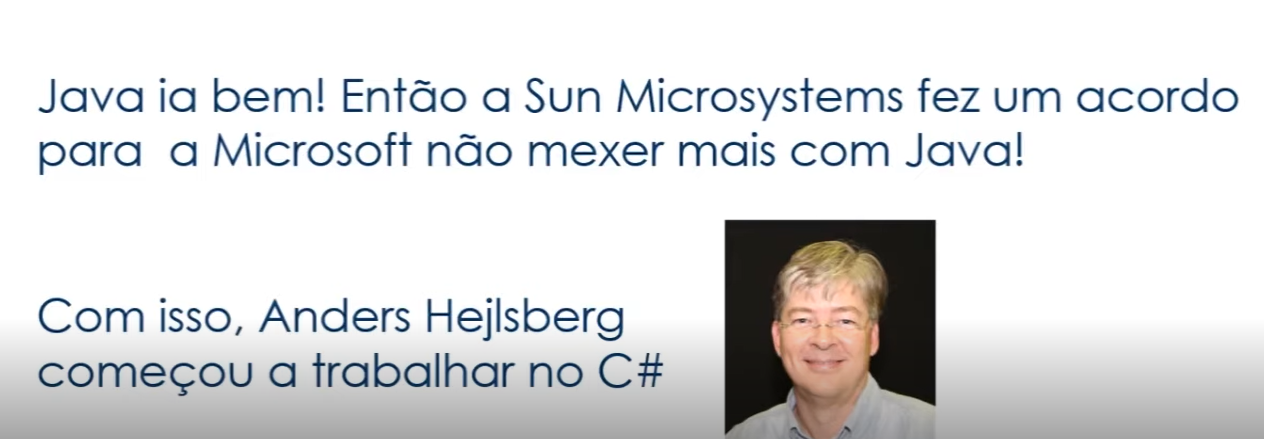
**Debugging**

- A depuração pode ser feita colocando um *BreakPoint* no canto esquerdo do código ou apertando F9 e executando ele

- F10 para ir depurando linha por linha e F5 para rodar todo o resto de uma vez

**TREINAMENTO 6: PRIMEIROS PASSOS COM .NET**

**1999:**



- O .NET foi lançado pela Microsoft em 2000

- Pensa que: cada linguagem tem um compilador que irá ser transformada em uma linguagem intermediária e então uma CLR irá transformar em linguagem de máquina

- A Microsoft melhorou a CLR 2 e então esta linguagem intermediária sendo interpretada pela CLR foi o que permitiu que o ambiente .NET servisse para várias linguagens



- A partir de 2016, existem dois frameworks .NET: .NET Framework e o .NET Core

- O .NET Framework já está ultrapassado e deixou de ser evoluído. O .NET Core foi evoluído para .NET 5

**O Que é, como e onde usar .NET**

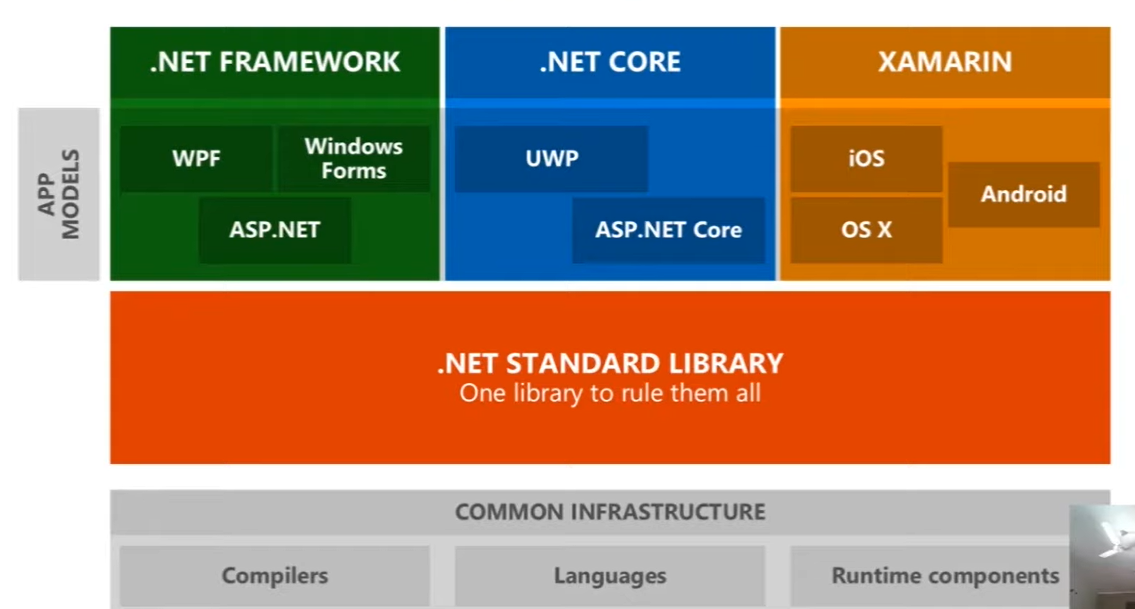
- Uma aplicação .NET é desenvolvida para e roda em uma das seguintes implementações do .NET:

.NET Core

.NET Framework

Mono

Universal Windows Plataform (UWP)



- Cada implementação inclui um ou mais .NET Runtimes (ambientes de execução)

- Atualmente a Microsoft desenvolve e suporta 3 linguagens para .NET: C#, F# e VB



**Conhecendo a CLI do .NET**

Dotnet –help

nuget: gerenciador de pacotes para .NET

**Conhecendo o C#**

- Os programas C# são executados no .NET, que inclui: CLR (Commom Language Runtime) e um Conjunto unificado de bibliotecas de classes.

- Atualmente o compilador do C# é o **Roslyn**

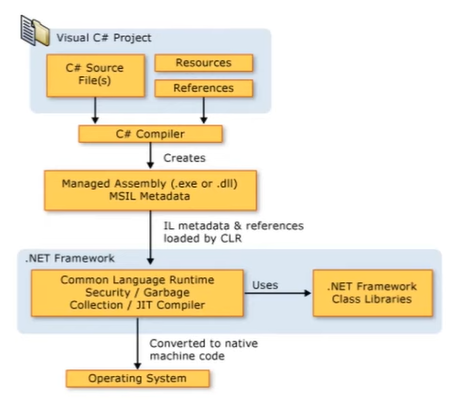
- O código fonte escrito em C# é compilado em uma linguagem intermediária (IL)

- O código e os recursos de IL são armazenados no disco em um arquivo executável chamado ***assembly*** (geralmente com uma extensão **.exe** ou **.dll**)

- Esses executáveis são o código compilado em uma linguagem intermediária

- Quando o programa C# é executado, o *assembly* (.exe ou .dll) é carregado no CLR

- Em seguida, **o CLR executará a compilação just in time (JIT) para converter o código IL em instruções de máquina nativas**



**Estrutura de programa**

- Principais conceitos organizacionais no C#:

Programas

Namespaces

Tipos

Membros

Assemblies

- Os programas são formados por um ou mais arquivos e declaram **TIPOS** que contêm **MEMBROS** e podem ser organizados em***NAMESPACES***

- Classes e interfaces são exemplos de tipos

- **Campos, métodos, propriedades** e **eventos** são exemplos de membros

**Tipos de valor**

- Existem dois tipos de variáveis: tipos de valor e de referência

- **Variáveis de tipos de valor** contêm diretamente seus dados: as variáveis têm sua própria cópia dos dados e não é possível que as operações afetem outra variável (exceto no caso das variáveis de parâmetro *ref* e *out*)

- Se eu clonar esta variável e fizer uma alteração no clone, isso não afetará a variável original porque cada uma tem a sua própria cópia na memória

|  |  |
| --- | --- |
| Numéricos | sbyte, short, int, long, byte, ushort, uint, ulong |
| Caracteres Unicode | char |
| Pontos flutuantes | float, double, decimal |
| Booleano | bool |
| Outros… | enum, struct, nullable |

- Variáveis de tipos de referência armazenam referências a seus dados: é possível que duas variáveis façam referência ao mesmo objeto...

- Nesse caso, as operações em uma variável afetam o objeto referenciado pela outra variável; tipos de referência:

|  |  |
| --- | --- |
| Tipos Classe | class, object, string |
| Tipos Arrays | Int[], int[,], etc... |
| Outros... | Interface, delegate |

**Instruções**

- As ações de um programa são expressas usando **INSTRUÇÕES,** geralmente delimitadas por um bloco

{

Instrução 1

Instrução 2

Instrução 3

}

- Exemplos: declaração de variáveis e constantes locais, condicionais (if, switch), repetição (for, while, do, foreach)

- Exemplos de instruções auxiliares: break, continue, return, tratativas de exceção (throw, try… catch… finally), using

**Exercícios práticos**

- **Criando um *if* que conta os argumentos de um array de string**

static void InstrucaoIf(string[] args)

{

if (args.Length == 0)

{

Console.WriteLine("Sem argumentos.");

}

else if (args.Length == 1)

{

Console.WriteLine("Um argumento");

}

else

{

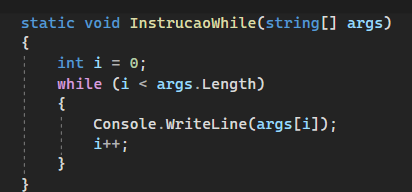
Console.WriteLine($"{args.Lenght} argumentos.");

}

}

- Podemos usar uma **INTERPOLAÇÃO DE STRINGS** que nem no Python usando o $”{}” na string

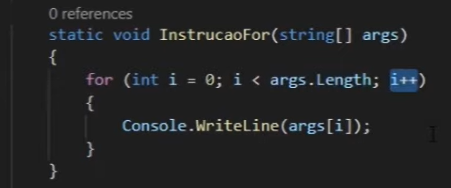
- While



**-Trabalhando repetição usando o do:** enquanto com o while a gente verifica a condição antes de executar o código, com o do essa verificação acontece no final

- Ou seja, o código será executado uma vez e haverá a verificação

**-For e foreach**



**Arrays**

- Estrutura de dados que contém um número x de elementos, **todos do mesmo tipo**, que são acessados através de índices computados []

- São tipos de referência e a declaração de uma **variável *array*** simplesmente reserva espaço para uma referência de uma instância de *array*

- Os índices de um array variam de 0 a n-1 ou comprimento do array -1

**Array unidimensional (lista)**

Int[] a = new int[10];

for (int i = 0; i < a.Length; i++)

{

a[i] = i \* i

}

For (int i = 0; i < a.Length; i++)

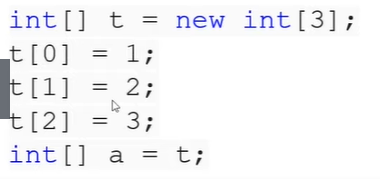
{

Console.WriteLine($”a[{i}] = {a [i]}”);

}

**Outras formas de inicializar um array**





**Array multidimensional (matriz)**

**Classes e Objetos**

**Classes**

- Tipo mais fundamental do C# e que permite a criação de objetos

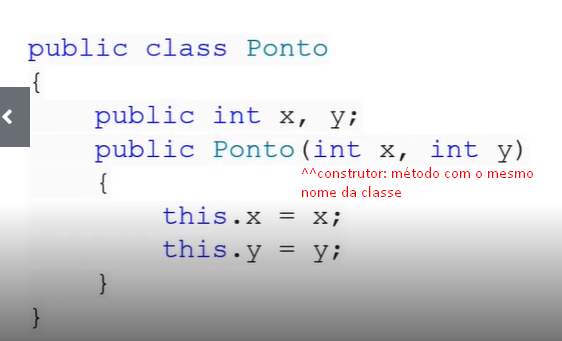
- Uma classe é uma estrutura de dados que combina estado (campos) e ações (métodos)

- A CLASSE é como se fosse a declaração da estrutura e, em tempo de execução, aquela classe será instanciada, criando um OBJETO e MEMÓRIA

**Objetos**

- São instâncias de uma classe

- As classes suportam **herança e polimorfismo**, mecanismos pelos quais as classes derivadas podem estender e especializar as classes base



- Toda vez que eu for criar um objeto naquela classe, tenho a opção de usar o construtor que defini

**Como criar um objeto?**

- São criados usando o operador new que aloca memória para uma nova instancia, chama um construtor para inicializar a instância e retorna uma referência à instância



- O p1 e p2 apenas referenciam a um objeto criado em algum lugar da memória

**Membros de uma Classe**

- Podem ser estáticos ou membros da instância (objeto)

- Membros estáticos pertencem à classe e membros de instância pertencem ao objeto

- Constantes, variáveis, métodos, propriedades, construtores, etc...

**Acessibilidade**

- Cada membro de uma classe tem uma acessibilidade associada que controla as regiões do texto do programa que podem acessar o membro

Podem ser:

public

protected (herança de uma classe base)

internal (só é acessado de dentro do assembly (csproject) que ele faz parte)

private (acessado única e exclusivamente dentro da classe em que ele está contido – nem mesmo de classes filhas)

**Herança**

- Uma declaração de classe pode especificar uma classe base, herdando os membros ***public, internal*** (se a classe filha estiver dentro do mesmo assembly da classe base) e ***protected*** da classe base

- Omitir uma especificação de classe base é o mesmo que derivar do tipo ***object***

**Métodos**

- É um membro que implementa uma computação ou ação que pode ser executada por um objeto ou classe

- Os métodos podem ter uma lista de parâmetros que representam valores ou referências de variáveis passados para o método...

- ... e um tipo de retorno que especifica o tipo do valor calculado e retornado pelo método

**Boa prática:** métodos devem ter nomes verbais (para indicar ação) e as propriedades ter nomes substantivos (para indicar características/propriedades)

**O que são Structs?**

- São estruturas de dados que podem conter membros de dados e membros de ação

- Mas, diferentemente das classes, as *structs* são tipos de valor (e não de referência) e não requerem alocação de *heap* (memória)

- Uma variável de um tipo de *struct* armazena diretamente os dados da estrutura, enquanto uma variável de um tipo de classe armazena uma referência a cum objeto alocado na memória

- Structs NÃO ACEITAM herança determinada pelo dev!! (diferentemente das classes)

- São úteis para pequenas estruturas de dados que possuem semântica de valor; exs.:

* Números complexos
* Pontos de um sistema de coordenadas
* Pares de chave-valor de um dicionário

- É uma estrutura simples!!!! A **vantagem** em relação a Classe é que usando *structs* estamos **alocando menos memória**

**Construtores**

- Construtores de *structs* são chamados com o operador **new**, semelhante a um construtor de classe...

- Mas em vez de alocar dinamicamente um objeto no *heap* gerenciado e retornar uma referência a ele,

um construtor *struct* simplesmente **retorna o próprio valor *struct*** (normalmente em um local temporário na *stack*)

e esse valor é copiado conforme necessário

! Não entendi esse lance de stack e heap !

<https://www.eximiaco.tech/pt/category/fundamentos>

**Entendendo interfaces e enums**

**Interfaces**

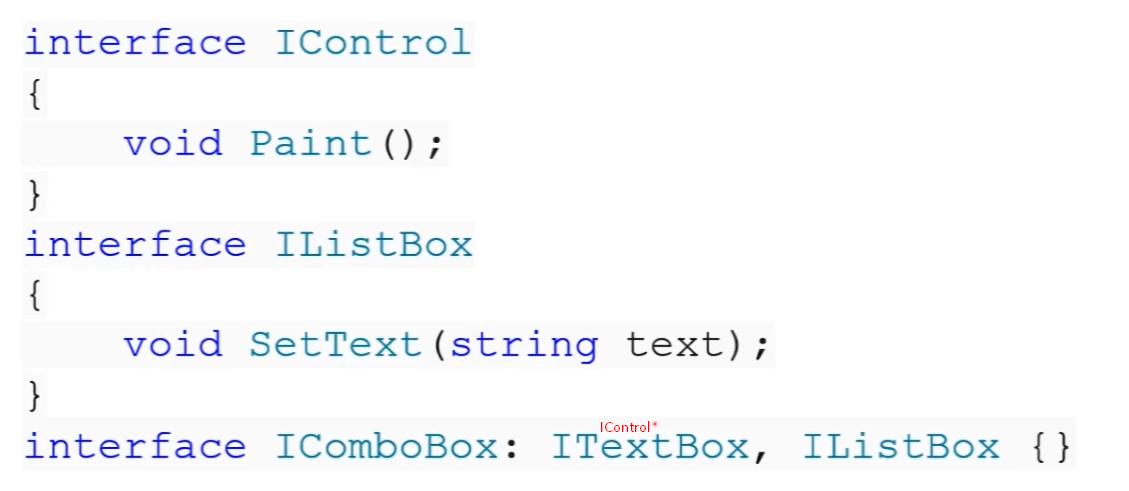
- É como se fosse um contrato

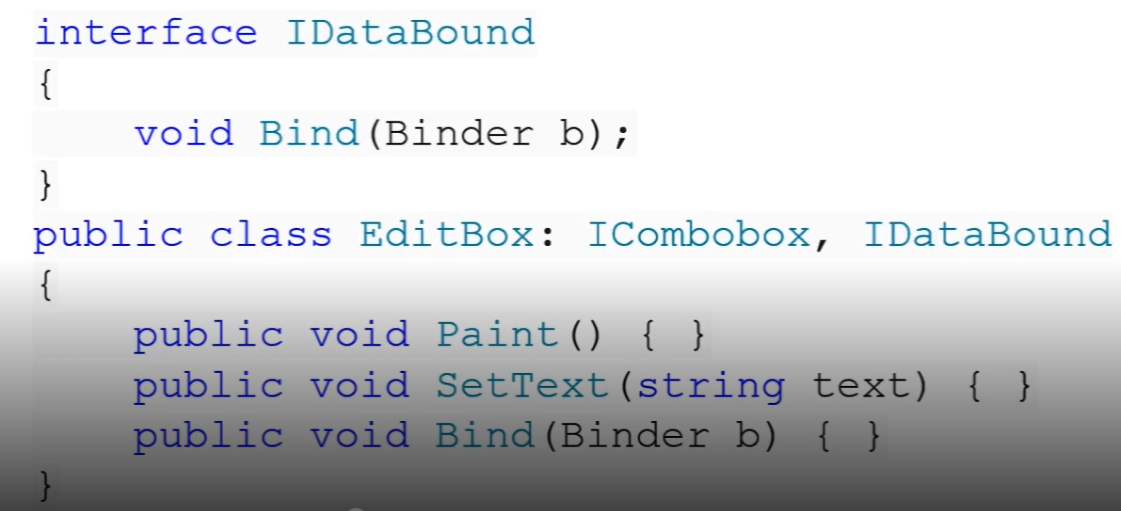
- Uma interface define um contrato que pode ser implementado por classes e structs

- Uma interface pode conter **métodos, propriedades, eventos e indexadores**

- Ela não fornece implementações dos membros e define apenas suas “assinaturas”

- As interfaces podem empregar herança múltipla

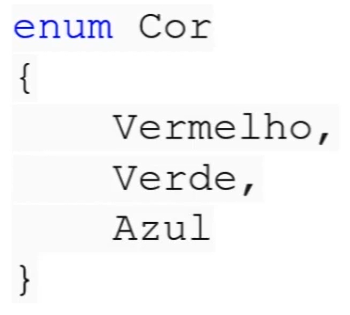




- Esse conceito é usado com muita frequência!!!

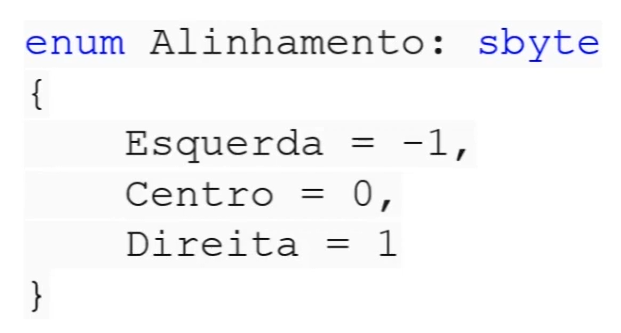
**Enums**

- É um tipo de valor distinto com um conjunto de constantes nomeadas



- Cada tipo de enum possui um tipo integral correspondente chamado tipo subjacente do tipo enum

- Um tipo de enumeração que não declara explicitamente um tipo subjacente tem um tipo subjacente *int*



- Se não informar que o tipo é sbyte, ele automaticamente assumiria que o tipo é int

**TREINAMENTO 7: PROGRAMAÇÃO ORIENTADA A OBJETOS COM C#**

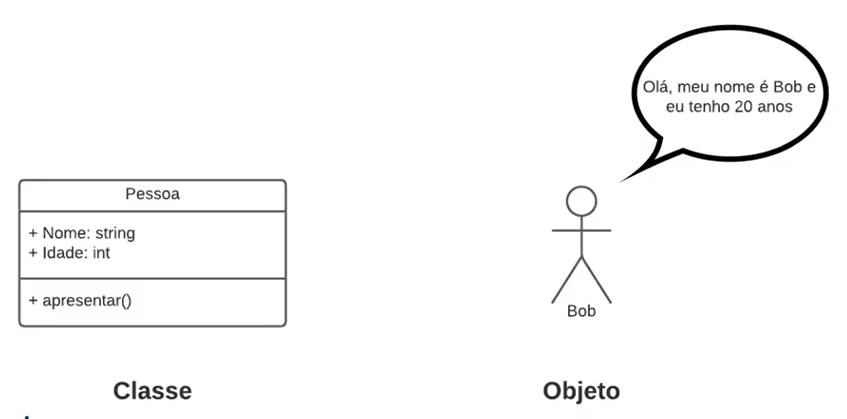
- A **POO é um paradigma** (metodologia) de programação, ou seja, corresponde a uma técnica de programação para um fim específico

Dentro desta técnica, existem quatro pilares:

* Abstração
* Encapsulamento
* Herança
* Polimorfismo

- Estes pilares se complementam e nenhum depende do outro, à exceção do polimorfismo que depende da herança

- Os principais conceitos dentro da POO são **classes** e **objetos**!



- A **classe**, nesse exemplo, é como se fosse um molde das características que precisamos daquela pessoa. Já o **objeto** é a pessoa em si, a sua materialização

- Pensa que podemos ter vários objetos do **tipo Pessoa** com base nessa classe do exemplo, certo?

**Tipos de paradigmas**

- É um conjunto de regras, técnicas e modelos para solucionar um problema

- Uma linguagem de programação implementa um ou mais paradigmas

Tipos existentes:

* POO
* Programação estruturada
* Programação imperativa
* Programação procedural
* Programação orientada a eventos
* Programação lógica
* Etc...

- Cada tipo implementa um conjunto específico de regras

- A POO se aproxima da vida real e torna muito mais fácil abstrair e solucionar um problema

**Abstração**

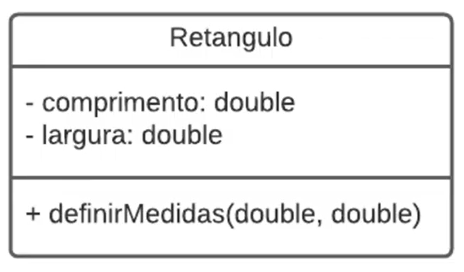
- Abstrair um objeto do mundo real para um contexto específico, considerando apenas os atributos importantes

**Encapsulamento**

- Serve para proteger uma classe e definir limites para alteração de suas propriedades

- Serve para ocultar seu comportamento e expor só o necessário





\*OBS.: para comentar um trecho de código tem que selecionar, apertar ctrl + K e SEM SOLTAR O CTRL apertar o C

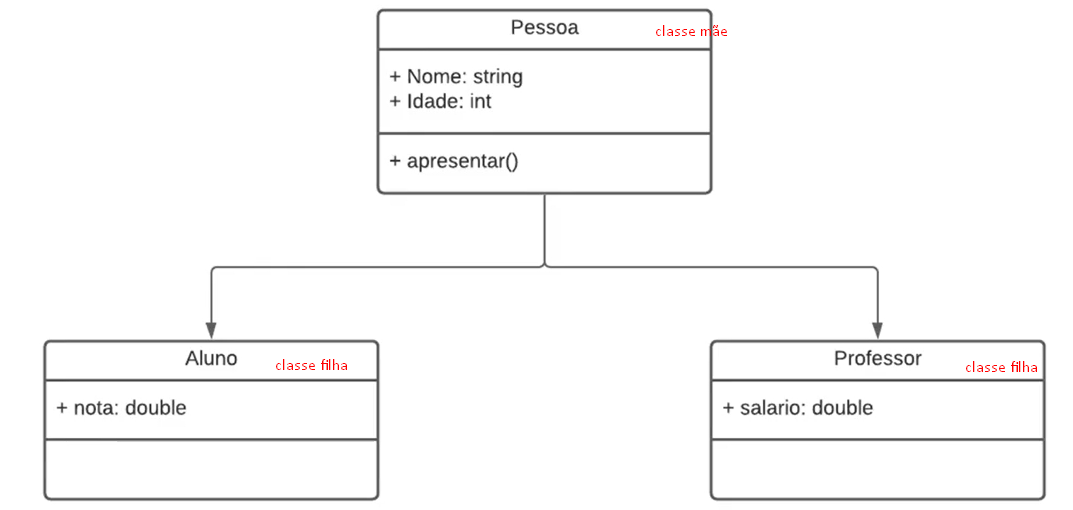
\*OBS2.: Por padrão, o valor de qualquer booleano é FALSO

**Herança**

- A herança nos permite reutilizar atributos, métodos e comportamentos de uma classe em outras classes

- Assim, nós temos classes derivadas de outras classes

- Serve para agrupar objetos que são do mesmo tipo, porém com características diferentes



- Assim, a classe filha herda os atributos da mãe, sem precisar ficar reescrevendo código

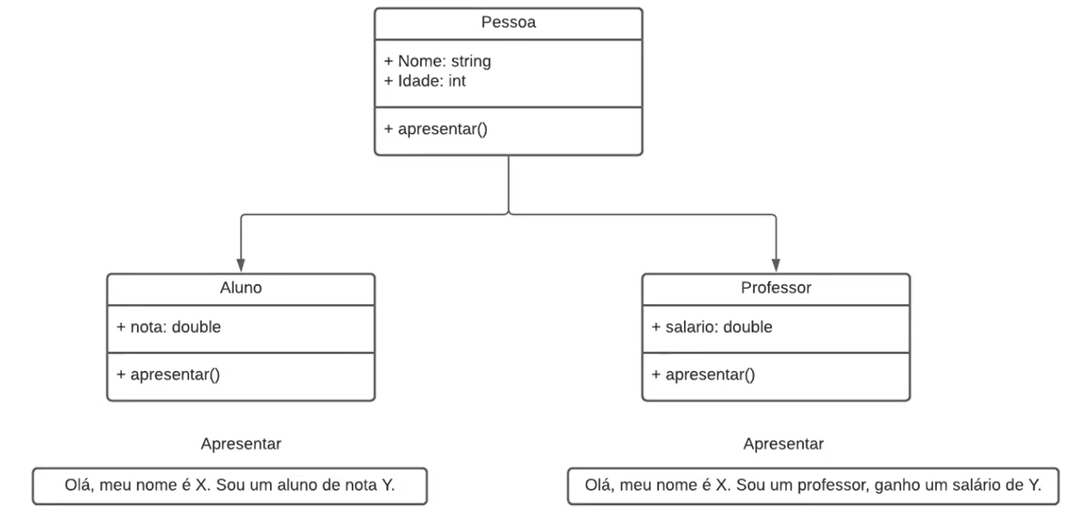
Ex.:



**Polimorfismo**

- “Muitas formas”

- Com o polimorfismo, podemos sobrescrever métodos das classes filhas para que se comportem de maneira diferente e ter sua própria implementação



- Existem dois tipos de polimorfismo

**Polimorfismo em tempo de compilação (Overload/Early Binding)**

- Muda apenas os parâmetros, mas o método é o mesmo



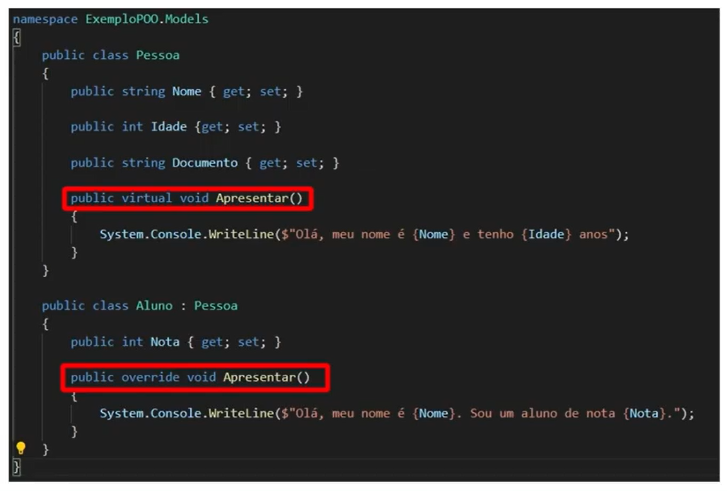
Lá no Program...



- A isso damos o nome de “sobrecarga de métodos”

**Polimorfismo em tempo de execução (Override/Late Binding)**

- Sobrescrita; repare o virtual e override



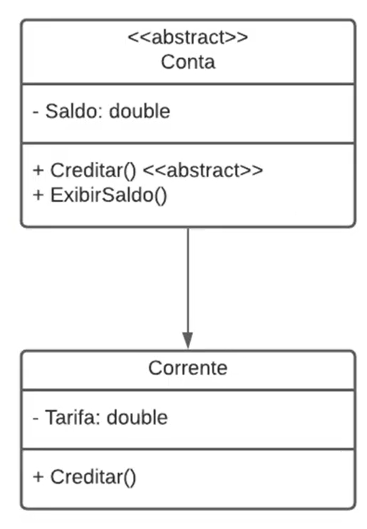
**Classes abstratas**

- Conceito associado à Herança

- Têm como objetivo ser, exclusivamente, um **molde** a ser herdado, portanto NÃO PODE ser instanciada

- É possível implementar métodos ou deixa-los a cargo de quem herdar

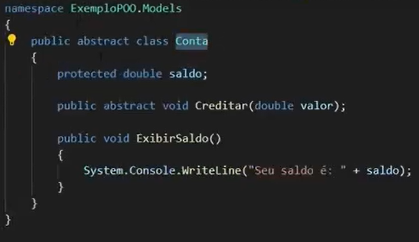
- Ela só tem função SE, e apenas se, for herdada



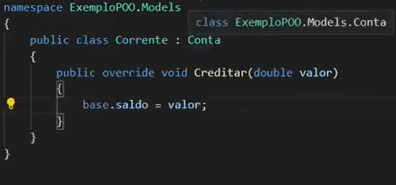
- Nesse caso, a classe filha Corrente é **OBRIGADA** a sobrescrever com polimorfismo o método Creditar() para que ele funcione

- Isso porque este é um método ABSTRATO

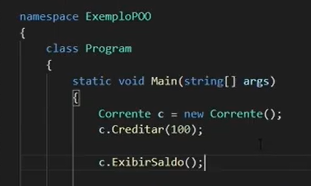
- Por outro lado, nesse caso é possível a classe filha usar o método ExibirSaldo() sem sobrescreve-lo (opcional)



Perceba aqui o override obrigatório de Creditar()



No meu Program...

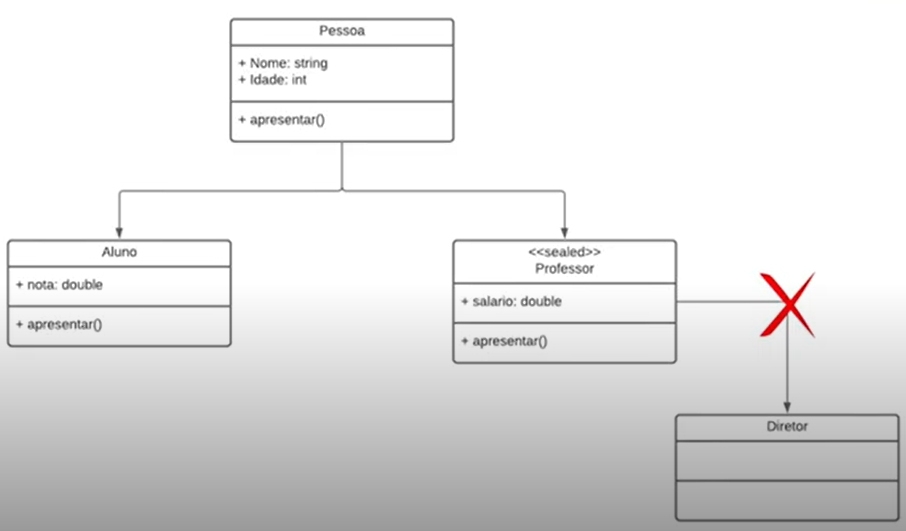


**Classes selada**

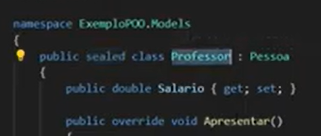
- Tem o objeto de impedir que outras classes derivem/façam uma herança dela

- Ou seja, mamãe não quer ter filho

- Também existem métodos e propriedades seladas



- Veja bem, uma classe selada pode receber herança de sua classe mãe, mas a sua classe mãe nunca será vovó



**Classe object**

- A classe System.Object é a mãe de todas as classes na hierarquia do .NET

- TODAS as classes derivem, direta ou indiretamente da classe Object e ela tem como objetivo prover serviços de baixo nível para suas classes filhas

- Oferece referência de memória:



- Nesse caso, a herança dos métodos ocorre de maneira implícita

**Interfaces**

- Uma interface é um contrato que pode ser implementado por uma classe

- Muito semelhante a uma classe abstrata, podendo definir métodos abstratos para serem implementados

- Assim como uma classe abstrata, uma interface não pode ser instanciada

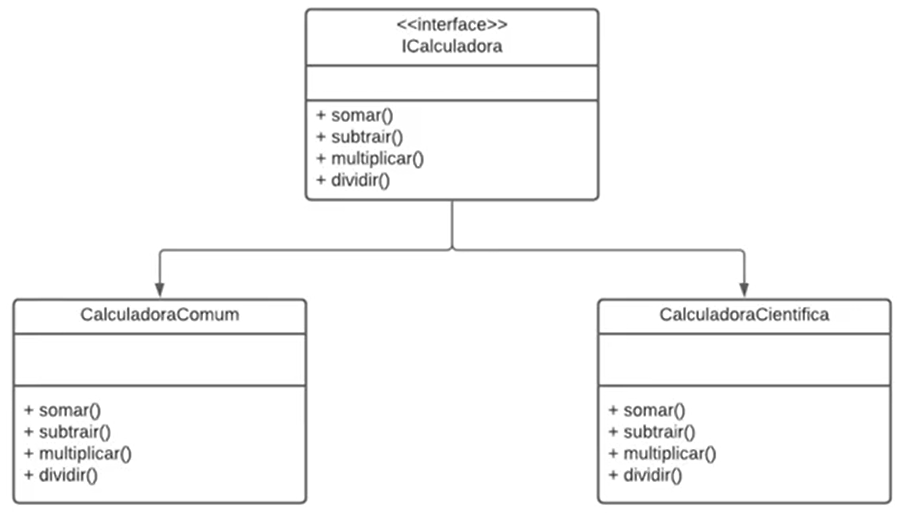
- Ou seja, a classe que usar, é obrigada a implementar com return etc o método que não foi implementado

- Uma classe pode implementar MÚLTIPLAS interfaces

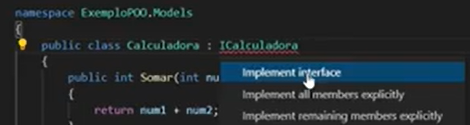
- Vamos lá: antes, se eu fizer um “contrato” pra utilizar uma interface, eu sou OBRIGADA a implementar todos os seus métodos

- agora, é possível colocar um retorno return padrão para os métodos da interface, e assim n vou precisar implementar obrigatoriamente na classe

- massss, caso esse retorno não exista na interface, aí sim sou obrigada a implementar na classe



- Por padrão, ao criar uma interface colocamos o I no começo do nome

- Para implementar a interface e corrigir o erro, clica Ctrl + . e clica em “Implementar Interface”

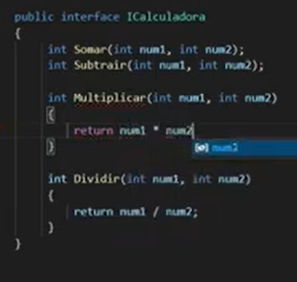


- Ele implementa vazio com exceção pq não sabe ainda qual a sua intenção ao implementar aquilo

- A dev que precisa escrever ali na classe o que cada método deve fazer



Exemplo de implementação na interface:

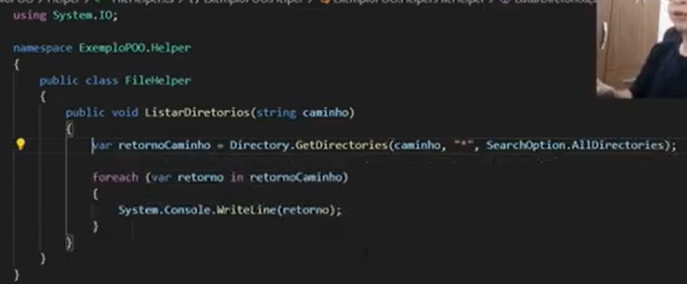


**Manipulação de Arquivos**

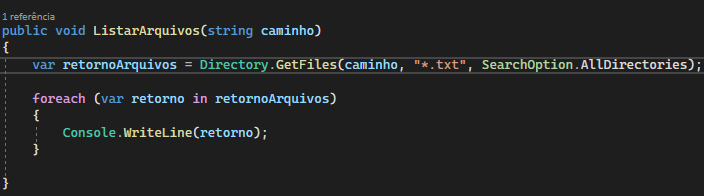
- O C# apresenta algumas classes estáticas que facilitam o trabalho com arquivos, dentre elas:

* File: manipulação de arquivos;
* Directory: operações com diretórios;
* Path: organização de caminhos de maneira lógica;

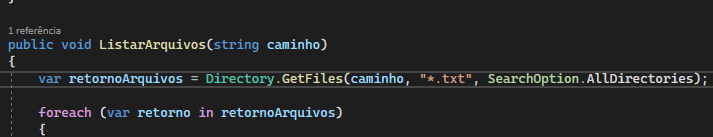
- É possível listar todas as pastas dentro das pastas do diretório que eu passar com a sobrecarga de métodos



**Listar arquivos**



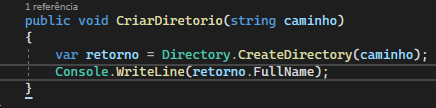
- Para selecionar um tipo específico de arquivo



-Para selecionar pelo nome, independentemente do tipo do arquivo



**Criar diretório**



- Usando o Path.Combine depois de importar o System.IO

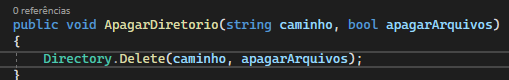


- Perceba ali em cima que ele tá criando uma pasta e subpasta

**Apagar diretório**

- Só é possível apagar um diretório pelo comando Directory.Delete caso este diretório esteja VAZIO

- Caso contrário, é preciso deixar claro que vc quer apagar o diretório E seus arquivos. Para isso, inserir um bool:



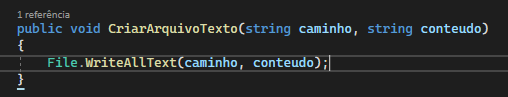
E no Program colocar true:



- Lembrando que ao deletar um arquivo pelo C#, ele NÃO VAI para a lixeira, ele é excluído de vez do pc

**Criar arquivo de texto**

- Cria o método



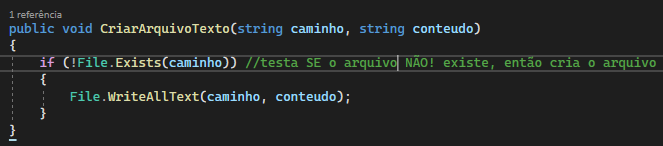
- Declara o caminho e o nome do arquivo



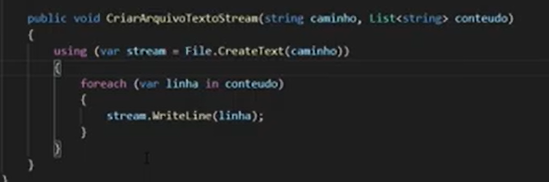
- Chama o método no Program



- Pode ser interessante testar se o arquivo já existe para que o C# não sobrescreva-o

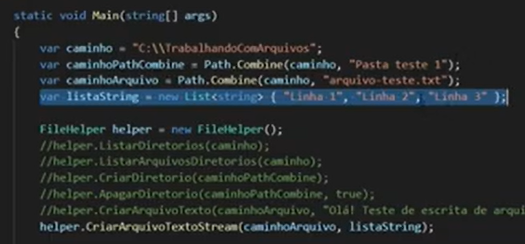


**Criando arq de texto com stream**



- O using serve para parar o processo quando a stream terminar de colocar as strings no arquivo

- Na classe Program...

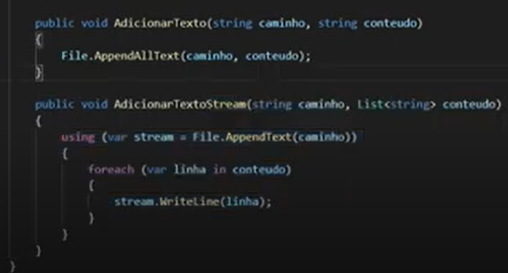


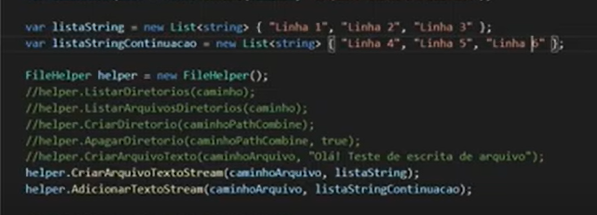
Passos:

* Criar a stream
* Fazer um laço
* Colocar o using

- Esta é a melhor maneira de trabalhar arquivos grandes

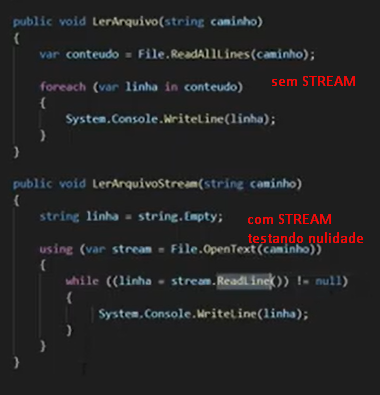
**Adicionar novas linhas**



No Program...

**Leitura do arquivo de texto**

- Com stream, novamente, é mais recomendado para arquivos grandes



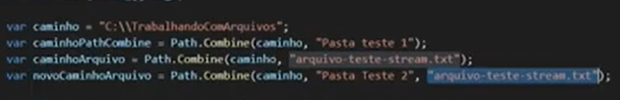
**Mover arquivo**

- No método tem que passar o caminho antigo e o novo caminho em que queremos que ele fique

- Além disso, n necessariamente precisamos mover ele com o mesmo nome



Lá no Program...



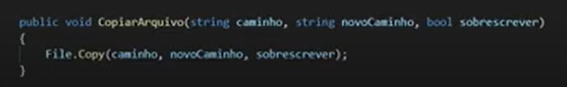


**Copiar arquivo**

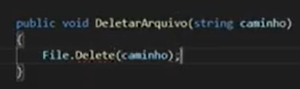
- Por default, o c# não permite copiar o arquivo para um diretório em que já exista um arquivo de mesmo nome

- Para contornar isso, é possível usar uma sobrecarga no método com um bool overwrite;

Assim, quando for no Program se colocar true, ele vai sobrescrever o arquivo mesmo que na pasta nova já exista um arquivo de mesmo nome



**Deletar arquivo**



**TREINAMENTO 7: CONSTRUTORES, PROPRIEDADES, DELEGATES E EVENTOS EM .NET**

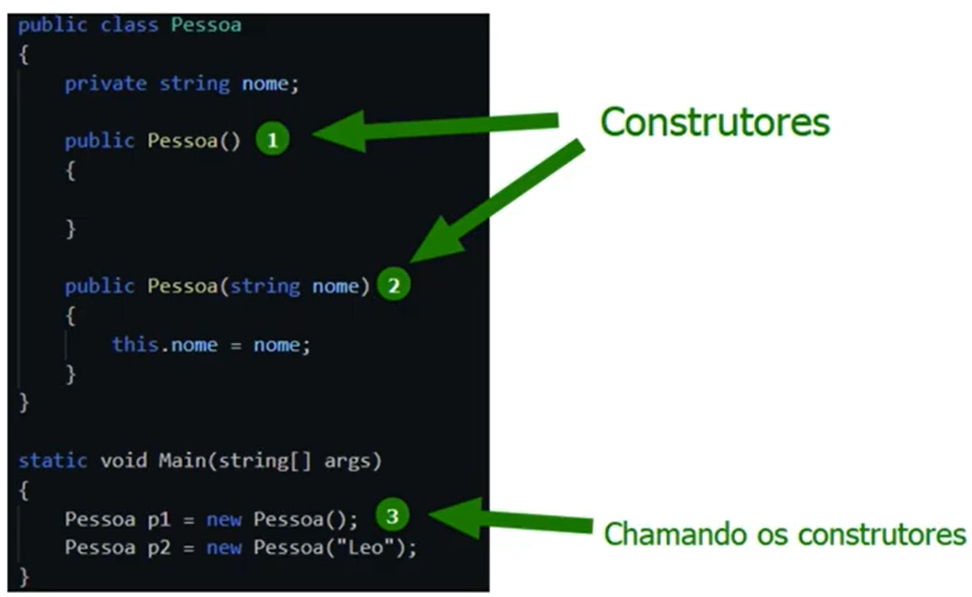
**Construtores**

- Um construtor é um método especial que contém o mesmo nome do seu tipo de classe

- Tem o objetivo de definir valores padrão, limitar uma instância e facilitar a instanciação de um objeto

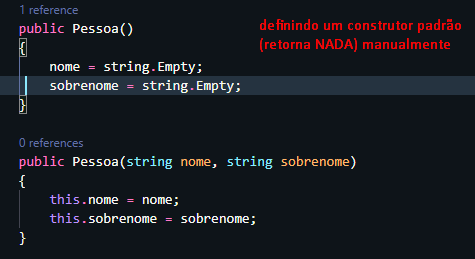
Um construtor...

* Não possui um retorno
* Um construtor padrão é sempre definido quando não declaramos nenhum para a sua classe
* Uma classe pode ter mais de um construtor



- ctor + tab é o atalho para a criação “automática” de construtor

- Se eu definir ao menos UM construtor para a minha classe, significa que a minha classe não tem mais um construtor padrão, a não ser que eu o defina manualmente



- Ou seja, para eu instanciar uma classe que defini ao menos UM construtor, OBRIGATORIAMENTE precisarei passar os seus parâmetros

**Construtor privado**

- Não é possível instanciar uma classe com um construtor privado

- Quando usamos o padrão de projeto **singleton**, podemos usar um construtor privado para bloquear a instância de uma classe

**Relembrando...** SDK é o conjunto de bibliotecas que vai fazer o pc desenvolver, entender e executar o meu projeto.