

SUMÁRIO

1 - PROGRAMAÇÃO ORIENTADA A OBJETOS

2- CLASSES

3- ATRIBUTOS

4-MÉTODOS

5 - TIPOS DE VARIÁVEIS

6- NÍVEIS DE ACESSO

7 - ENCAPSULAMENTO

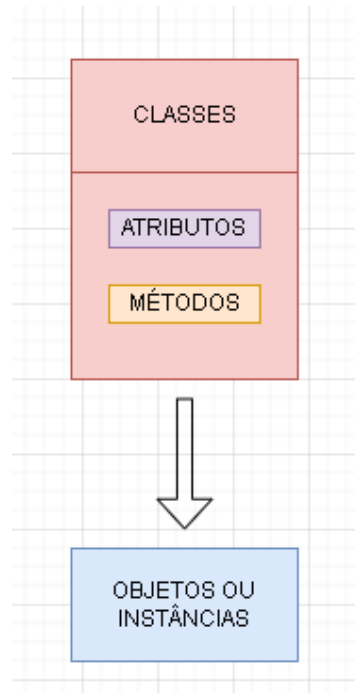
8 - AGREGAÇÃO

9 - HERANÇA / POLIMORFISMO

PROGRAMAÇÃO ORIENTADA A OBJETOS

A programação orientada a objetos consiste resumidamente em uma maneira de você organizar o seu código.

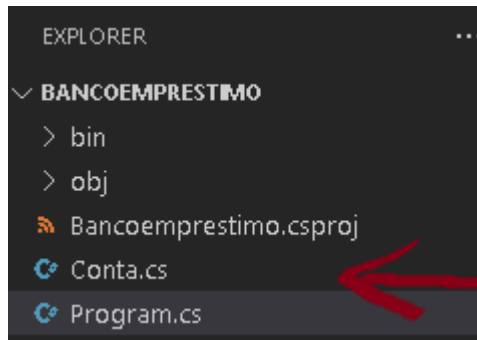
Na maneira como nós aprendemos, ela funciona na seguinte estrutura:



Para explicar essa estrutura, irei utilizar o programa que criamos que simula uma conta bancária.

CLASSE

É um arquivo do nosso programa onde colocaremos os atributos e métodos que iremos utilizar.



No caso a classe que estamos utilizando para declarar essas informações é a “Conta.cs” e na classe program.cs, que é a classe principal nós iremos instanciar(criar) um objeto para essa classe, pois nele, nós iremos adicionar valores para esses atributos e argumentos para nossos métodos

```
namespace banco;
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Banco Etec MCM");
        Conta c = new Conta();
        c.Depositar(1500);
        c.AjustaLimite(500);
        c.Sacar(500);
        Console.WriteLine("Seu saldo é de: " + c.MostrarSaldo());
    }
}
```

ATRIBUTO

São as informações que desejamos atribuir a nossa classe, elas seguem o formato: “nível de acesso, tipo, nome”;

```
public int Numero {get; set;}  
4 references  
private double Saldo {get; set;}  
  
3 references  
public double Limite {get; private set;}  
  
1 reference  
public void Depositar(double valor){  
    this.Saldo += valor;  
}
```

MÉTODO

São ações que nós desejamos que nossas classes realizem, elas seguem o formato: “nível de acesso, tipo, nome(argumento);”

```
1 reference
public void Sacar(double valor){
    if(valor > this.Saldo + this.Limite) {
        Console.WriteLine("Você não pode realizar essa operação, o saldo é insufi
    } else{
        this.Saldo -= valor;
    }
}
```

É importante ressaltar que na declaração tanto dos atributos, quanto dos métodos na nossa classe precisamos definir o seu nível de acesso e qual é o tipo dela.

TIPOS DE VARIÁVEIS

Como eu citei os atributos, os métodos e os argumentos, precisam receber valores que serão utilizados para realizar o propósito do programa, ele é a segunda coisa que escrevemos ao declarar atributos e métodos, e a primeira nos argumentos, por exemplo:

```
public int numero
```

```
public void sacar(double valor)
```

Os tipos de variáveis que normalmente usamos nas aulas para armazenar valores são:

INT = NÚMEROS INTEIROS

DOUBLE = NÚMEROS DECIMAIS

STRING = CARACTERES

BOOL - BOOLEANO (VERDADEIRO OU FALSO)

VOID - NULO

A variável VOID utilizamos quando o nosso atributo ou o nosso método, não precisa retornar para o usuário algum valor, por exemplo:

```
1 reference
public void Sacar(double valor){
    if(valor > this.Saldo + this.Limite) {
        Console.WriteLine("Você não pode realizar essa operação, o saldo é insufi
    } else{
        this.Saldo -= valor;
    }
}
```

No nosso programa do banco, o método de sacar, serve apenas para determinar se a operação de saque pode ou não ser realizada de acordo com o dinheiro que você tem de saldo, mas em momento algum no programa ele retoma algum valor, sendo ele um número ou um caractere por exemplo, ele apenas retorna uma mensagem, caso não seja possível.

```
1 reference
public double MostrarSaldo(){
    return this.Saldo + this.Limite;
}
```

Nesse método de mostrar saldo, perceba que o tipo é double, e no final ele tem como objetivo retornar o valor do nosso saldo + o valor do nosso limite, realizando o retorno de um valor do tipo double

```

Banco Etec MCM
Seu saldo é de: 1500
O valor do seu empréstimo total é de: 1719,9999999999998
PS C:\Users\unkno\OneDrive\Área de Trabalho\AulasSW-main\Bancoemprestimo>

```

NÍVEIS DE ACESSO

Podemos definir níveis de acesso para os nossos atributos e métodos, eles são a primeira coisa que escrevemos ao declarar uma variável (atributos ou métodos no caso)

Normalmente utilizamos **Public** (público), porém o **Private** (privado) serve para algumas situações.

```

0 references
public int Numero {get; set;}
4 references
private double Saldo {get; set;}

```

Vamos supor que eu possuo um banco com o Guilherme e eu tenha dois clientes nesse banco, a Julia e o Victor, todo final do mês eu vou lá na “Conta.cs” e deposito o salário deles na conta,

```
c.Depositar(1500);
```

Só que vamos supor que o Guilherme é muito fã da Julia e ele decida entrar na “program.cs” e adicionar 1 milhão para a Julia diretamente no saldo dela e altere o valor:

```
c.saldo(1000000000000);
```

Para impedir que esse tipo de coisa aconteça e o programa se corrompa, nós podemos tornar o atributo privado para que a pessoa não tenha acesso a edição dele, como foi feito com o saldo nesse programa, impossibilitando que essa ocasião acontecesse.

Além disso outra opção para resolver esse problema seria o encapsulamento ({get; set;})

ENCAPSULAMENTO

O encapsulamento usa do get (método de leitura) e o set (método de gravação) para prever que os valores de um atributo só possam ser alterados diretamente na classe que eles são atribuídos

```
4 references  
private double Saldo {get; set;}  
  
3 references  
public double Limite {get; private set;}
```

O atributo "Saldo" é definido como privado na classe "Conta". Isso significa que seu valor não pode ser acessado diretamente de fora da classe. Em vez disso, é fornecido um método de leitura (get) e um método de gravação (set) para controlar o acesso a esse atributo.

O método get da propriedade "Saldo" permite obter o valor atual do saldo. Ele é usado para recuperar e retornar o valor do atributo "Saldo". Já o método set permite atribuir um novo valor ao saldo. É importante notar que, neste exemplo, o set não é exposto publicamente. Isso significa que o saldo só pode ser modificado internamente, através de métodos dentro da classe "Conta".

Atributo "Limite":

O atributo "Limite" também é definido como privado na classe "Conta". No entanto, ele possui um set privado e um get público.

O set privado permite que a classe "Conta" controle a atribuição de um novo valor ao limite. Somente métodos internos da classe podem modificar o limite chamando o método `AjustaLimite`. Isso permite que a classe aplique lógica adicional ou restrições ao definir o limite, caso necessário.

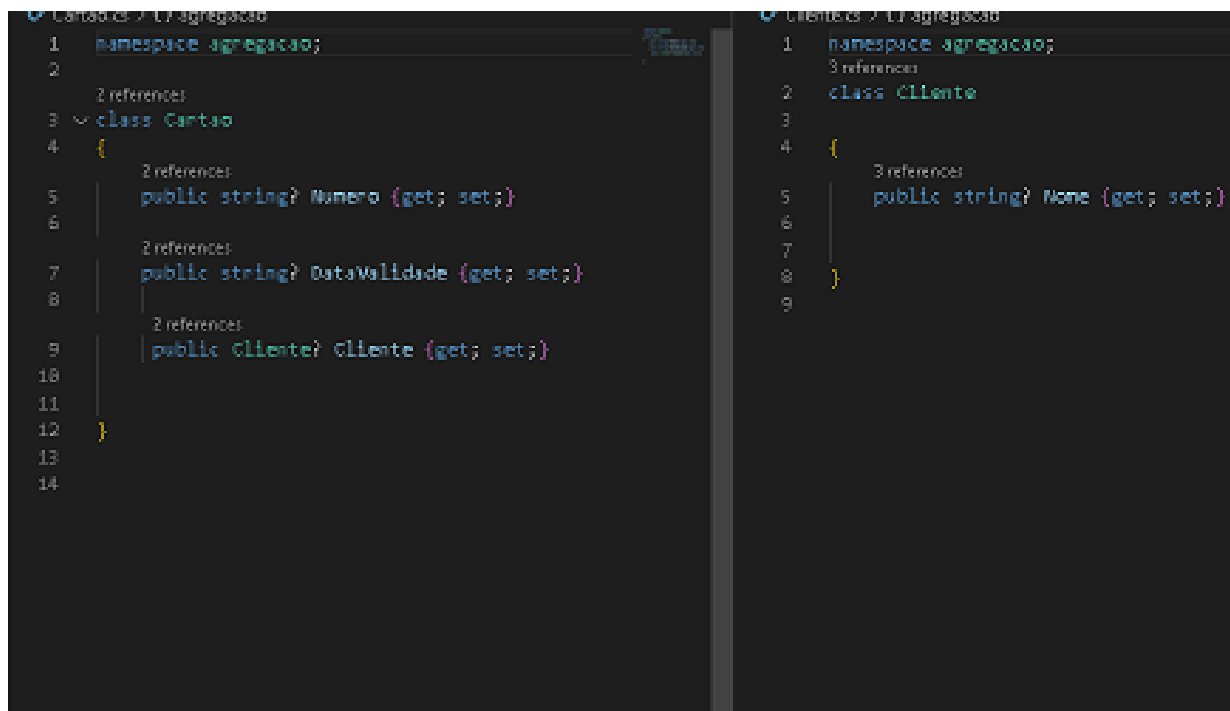
O get público permite que outros códigos externos leiam o valor atual do limite chamando o método `MostrarSaldo`. Assim, é possível obter o limite atual da conta, mas não é possível modificá-lo diretamente de fora da classe.

Resumindo, o uso do get e set nos atributos "Saldo" e "Limite" permite um controle mais preciso sobre o acesso e modificação desses atributos. O get permite ler os

valores, enquanto o set controla como esses valores são definidos, garantindo a consistência e a segurança dos dados dentro da classe "Conta".

AGREGAÇÃO

A agregação consiste basicamente no relacionamento dos itens de duas classes diferentes, para demonstrar irei utilizar um exercício



Neste exercício, onde programamos um cartão de crédito, observe que a classe nome interage diretamente com o atributo cliente da classe cartão, em vista que o atributo Nome será o valor do Cliente, portanto, para que tal relação aconteça é preciso apenas interliga-las na classe principal.

```
Conta cdc = new Conta();
Cliente cli = new Cliente();

cli.Nome = "Nicolas Garcia";

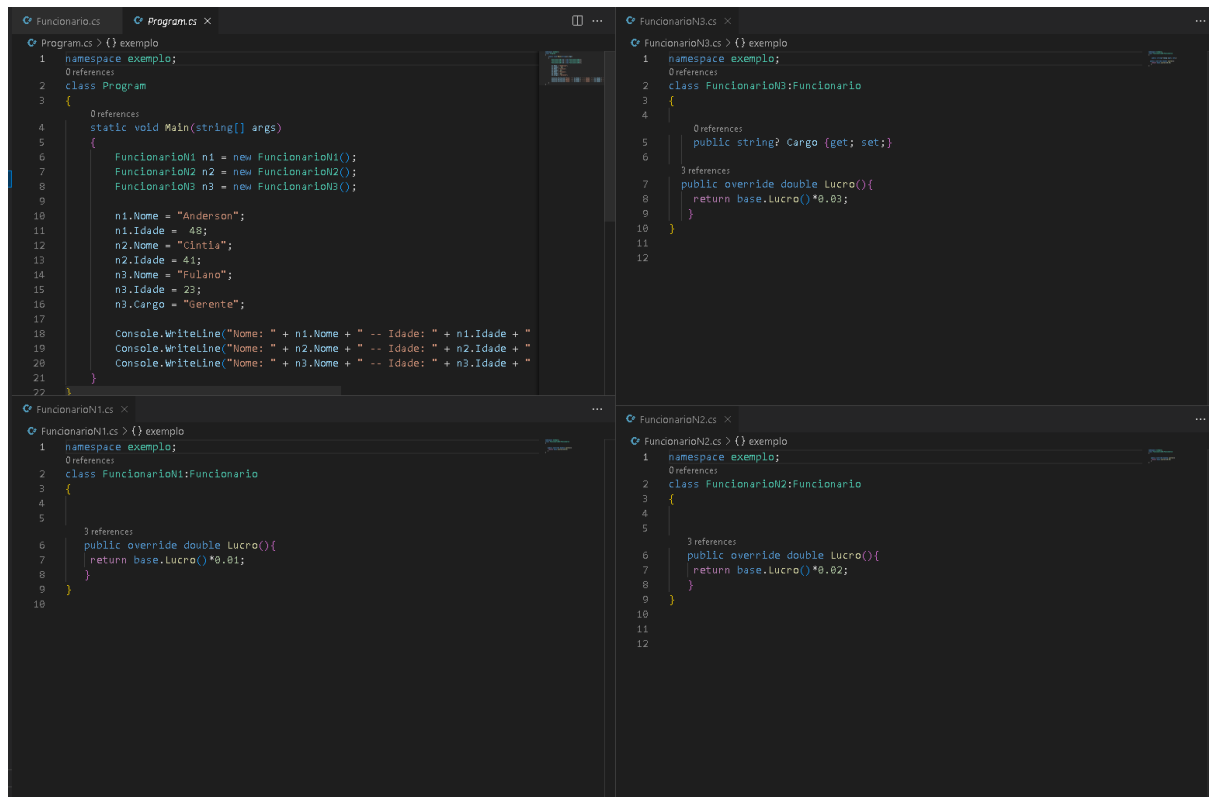
cdc.Numero = "509875017410712";

cdc.DataValidade = "08/2026";

cdc.Cliente = cli;
```

HERANÇA / POLIMORFISMO

Para exemplificar a herança e o polimorfismo, eu utilizarei esse exercício:



```
Program.cs > {} exemplo
1 namespace exemplo;
2 class Program
3 {
4     static void Main(string[] args)
5     {
6         FuncionarioN1 n1 = new FuncionarioN1();
7         FuncionarioN2 n2 = new FuncionarioN2();
8         FuncionarioN3 n3 = new FuncionarioN3();
9
10        n1.Nome = "Anderson";
11        n1.Idade = 48;
12        n2.Nome = "Clintis";
13        n2.Idade = 41;
14        n3.Nome = "Fulano";
15        n3.Idade = 23;
16        n3.Cargo = "Gerente";
17
18        Console.WriteLine("Nome: " + n1.Nome + " -- Idade: " + n1.Idade + "
19        Console.WriteLine("Nome: " + n2.Nome + " -- Idade: " + n2.Idade + "
20        Console.WriteLine("Nome: " + n3.Nome + " -- Idade: " + n3.Idade + "
21
22    }
23 }
```

```
Funcionario.cs > {} exemplo
1 namespace exemplo;
2 class Funcionario
3 {
4     public string? Cargo { get; set; }
5
6     public override double Lucro()
7     {
8         return base.Lucro() * 0.03;
9     }
10 }
```

```
FuncionarioN1.cs > {} exemplo
1 namespace exemplo;
2 class FuncionarioN1:Funcionario
3 {
4
5
6     public override double Lucro()
7     {
8         return base.Lucro() * 0.01;
9     }
10 }
```

```
FuncionarioN2.cs > {} exemplo
1 namespace exemplo;
2 class FuncionarioN2:Funcionario
3 {
4
5
6     public override double Lucro()
7     {
8         return base.Lucro() * 0.02;
9     }
10 }
```

A herança diz a respeito de uma classe herdar elementos de outra nesse caso, os outros três funcionários herdam o nome e a idade da classe mãe (funcionario), ao colocar “:nomedaclasseque será herdada” depois do nome da classe que herdará.

O polimorfismo diz a respeito sobre quando um elemento do programa pode assumir mais de uma forma, no caso, o lucro (que é derivado da classe mãe funcionário) de cada um deles, é diferente apesar de ele ser o mesmo método, já que cada classe possui um lucro diferente