

## O que é o Framework .NET?



O .NET Framework é um ambiente de execução criado pela Microsoft e gerenciado para Windows que oferece uma série de serviços voltados ao desenvolvimento web, reutilizando e reaproveitando códigos, entre suas principais funções.

É um ambiente outsource que possui componentes para a criação de códigos em determinadas linguagens, como C#, VB.NET e F#.

Pode-se dizer que o .NET Framework engloba dois componentes principais, como:

- CLR (Common Language Runtime): um mecanismo de execução que manipula aplicativos em execução;
- Biblioteca de classes: o .NET Framework oferece uma biblioteca de códigos testados e reutilizáveis que os desenvolvedores podem chamar de seus próprios aplicativos.

Uma primeira grande vantagem do .NET Framework é o poder de reutilização de estruturas de código.

Isso poupa horas e horas de desenvolvimento e faz com que os desenvolvedores possam focar no que é de fato importante e que agrega valor ao negócio com relação ao software que está sendo desenvolvido.

Ou seja, você não precisa dedicar tempo para desenvolver a funcionalidade de login, já que existem frameworks já testados para essa finalidade.

Ainda, se necessário, você pode personalizar esses componentes pré-disponibilizados de acordo com as demandas do projeto em questão.

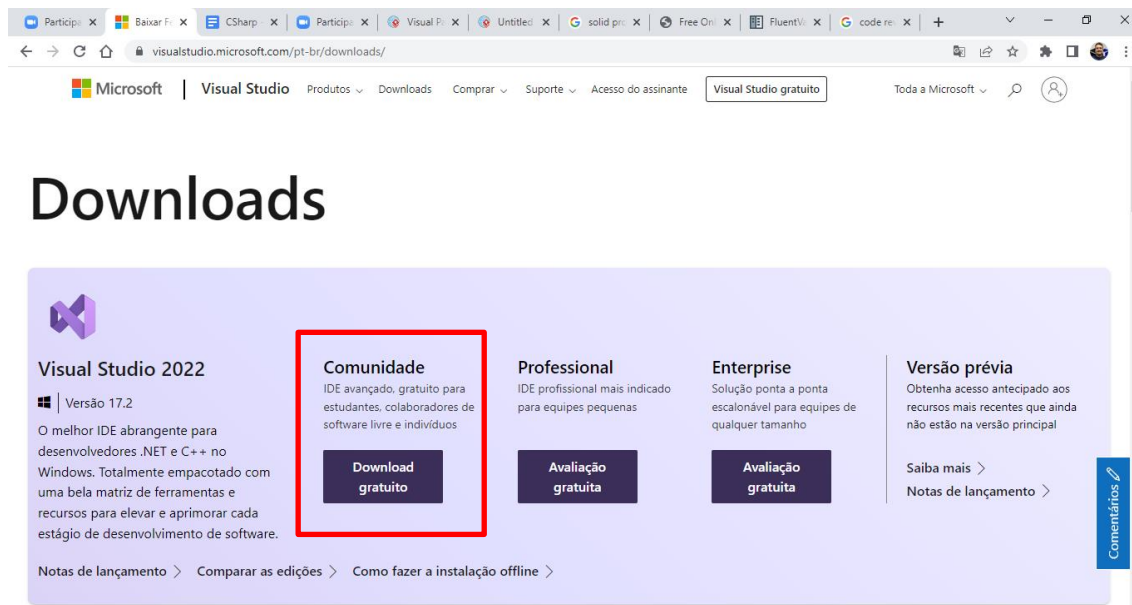
## Visual Studio 2022

IDE: Ambiente integrado de desenvolvimento.

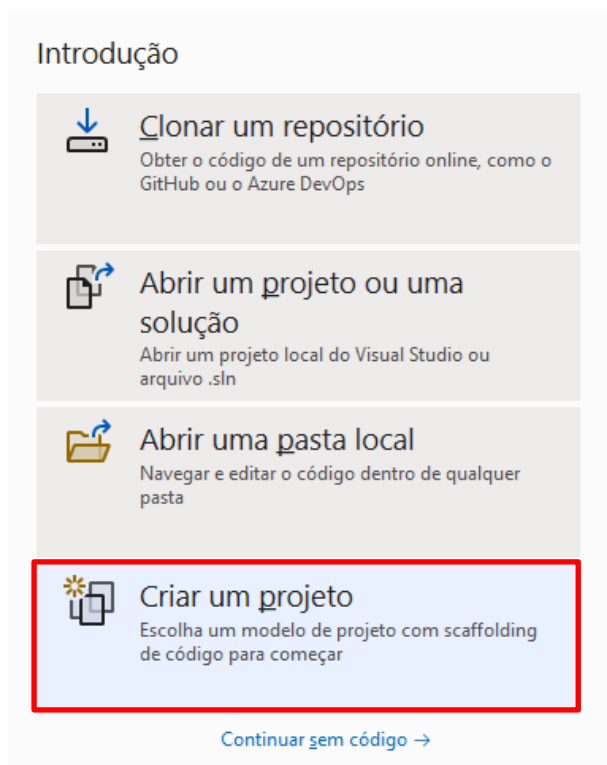
- Versão do .NET: 6
- Linguagem: C#

### Baixando o Visual Studio:

<https://visualstudio.microsoft.com/pt-br/downloads/>

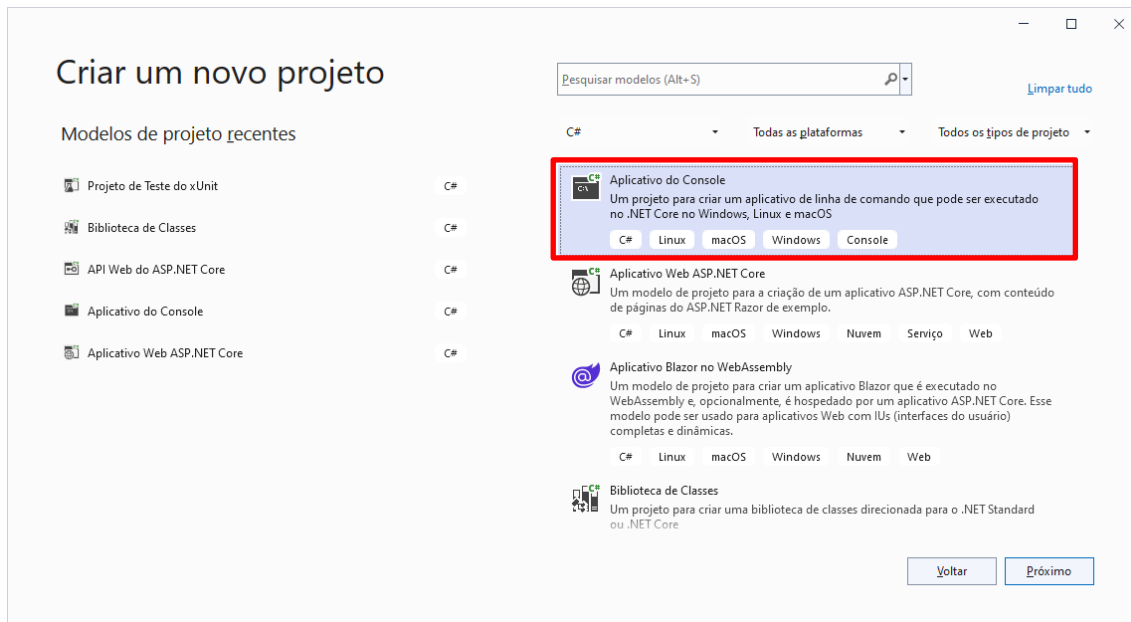


### • Criando um projeto:



## Aplicativo do Console

Tipo de projeto para desenvolvimento em modo DOS (Prompt de comando)



Criar um novo projeto

Modelos de projeto recentes

- Projeto de Teste do xUnit
- Biblioteca de Classes
- API Web do ASP.NET Core
- Aplicativo do Console
- Aplicativo Web ASP.NET Core

Aplicativo do Console

Um projeto para criar um aplicativo de linha de comando que pode ser executado no .NET Core no Windows, Linux e macOS

C# Linux macOS Windows Console

Aplicativo Web ASP.NET Core

Um modelo de projeto para a criação de um aplicativo ASP.NET Core, com conteúdo de páginas do ASP.NET Razor de exemplo.

C# Linux macOS Windows Nuvem Serviço Web

Aplicativo Blazor no WebAssembly

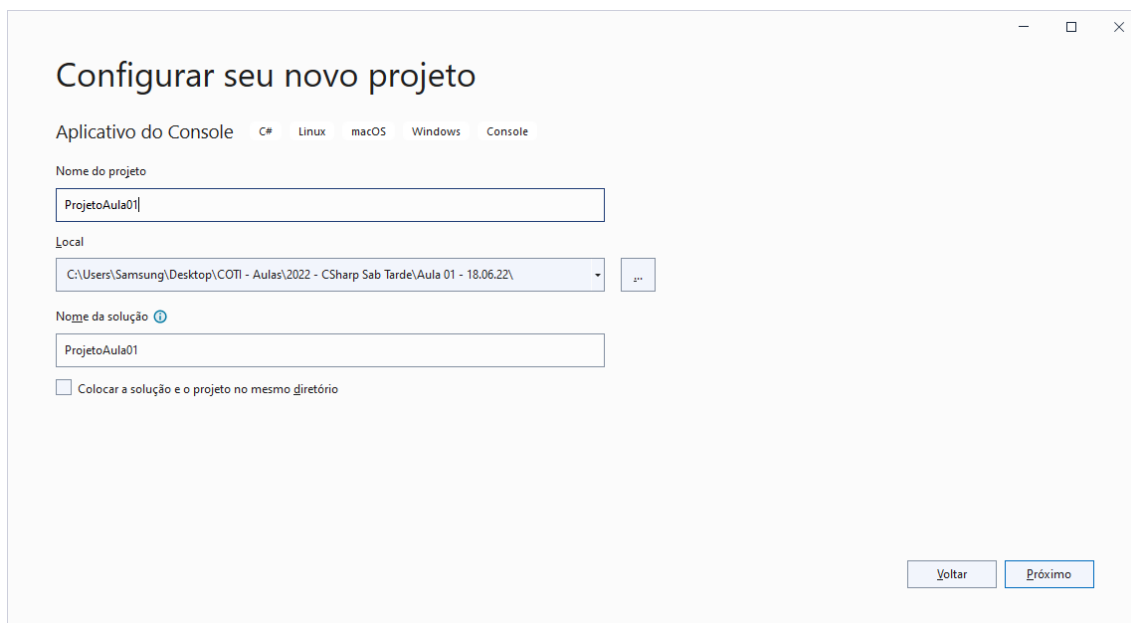
Um modelo de projeto para criar um aplicativo Blazor que é executado no WebAssembly e, opcionalmente, é hospedado por um aplicativo ASP.NET Core. Esse modelo pode ser usado para aplicativos Web com IUs (interfaces do usuário) completas e dinâmicas.

C# Linux macOS Windows Nuvem Web

Biblioteca de Classes

Um projeto para criar uma biblioteca de classes direcionada para o .NET Standard ou .NET Core

Voltar Próximo



Configurar seu novo projeto

Aplicativo do Console C# Linux macOS Windows Console

Nome do projeto

ProjetoAula01

Local

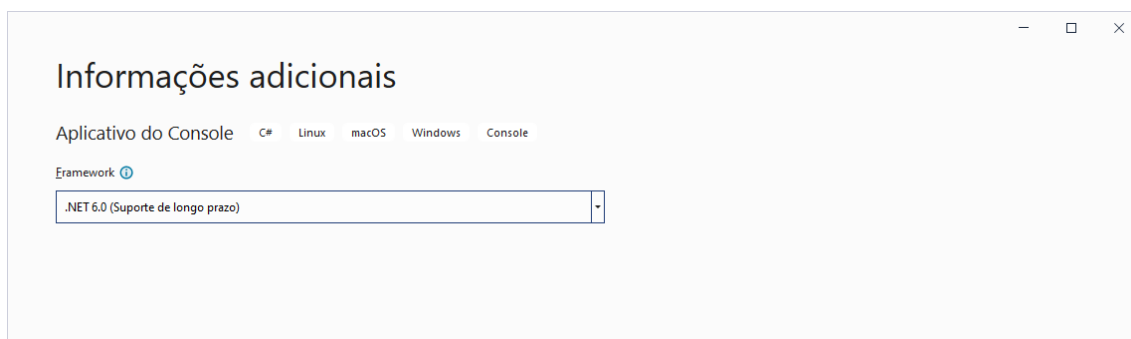
C:\Users\Samsung\Desktop\COTI - Aulas\2022 - CSharp Sab Tarde\Aula 01 - 18.06.22\

Nome da solução

ProjetoAula01

☐ Colocar a solução e o projeto no mesmo diretório

Voltar Próximo



Informações adicionais

Aplicativo do Console C# Linux macOS Windows Console

Framework

.NET 6.0 (Suporte de longo prazo)

## /Program.cs

Classe principal do projeto Console utilizada para executar o projeto.

```
//namespace -> localização da classe no projeto
namespace ProjetoAula01
{
    //declaração da classe
    //public -> visibilidade da classe (acesso total)
    public class Program
    {
        //método que tem a função de executar
        //o projeto Console
        public static void Main(string[] args)
        {
            //imprimindo mensagem no console
            Console.WriteLine("Aula 01 - C# WebDeveloper");

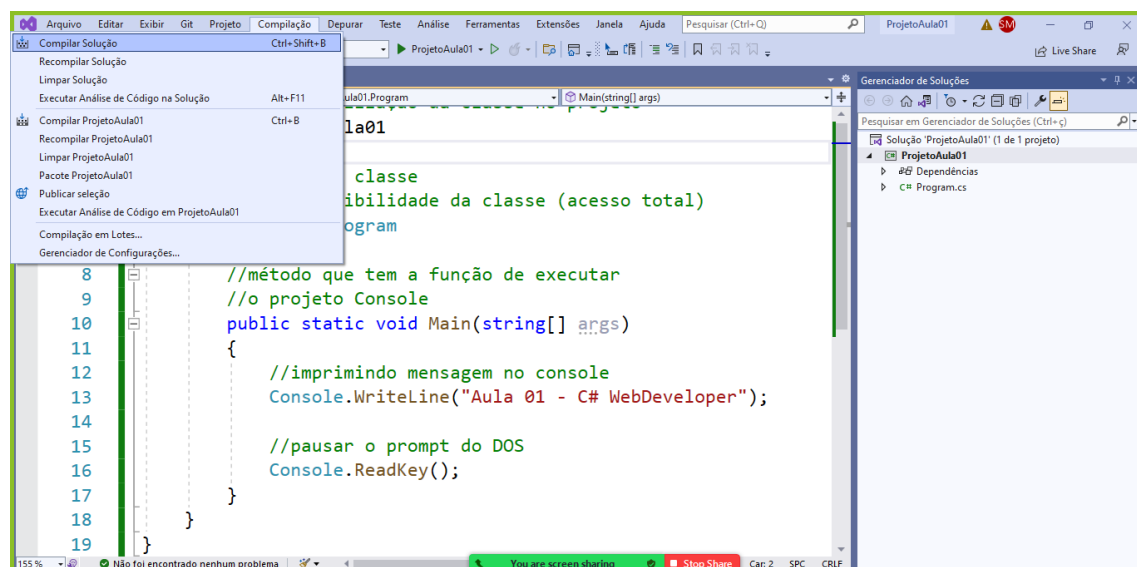
            //pausar o prompt do DOS
            Console.ReadKey();
        }
    }
}
```

Em C#, para imprimir os dados na tela de saída do console, os seguintes métodos são usados - método **Console.Write()** e **Console.WriteLine()**.

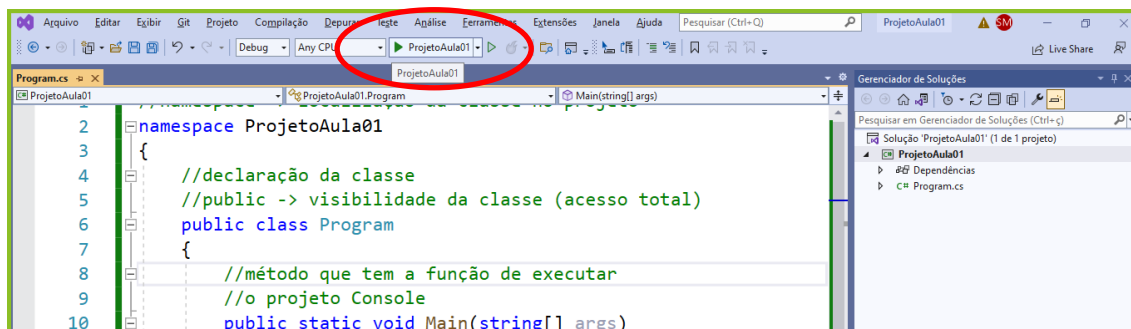
Console é uma classe predefinida de namespace System . Enquanto Write() e WriteLine() ambos são os métodos Console Classe.

A única diferença entre Write() e WriteLine() é que Console.Write é usado para imprimir dados sem imprimir a nova linha, enquanto Console.WriteLine é usado para imprimir dados junto com a impressão da nova linha.

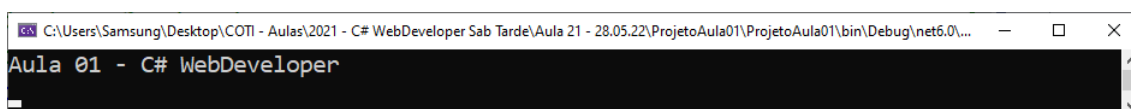
## Compilando o projeto:



**Executando:** Tecla de atalho: F5



**Saída do programa:**



**Tarefa:**

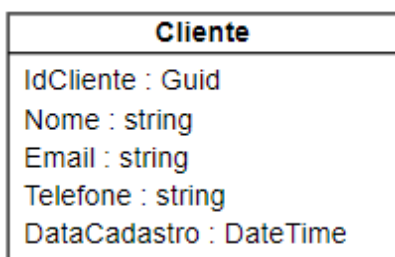
Criar um projeto para exportação de dados de clientes para arquivos. O sistema deverá funcionar da seguinte maneira:

- O sistema deverá pedir ao usuário que preencha os dados de um cliente, composto de:
  - Nome
  - Email
  - Telefone
  - Data de Cadastro
- O Sistema deverá armazenar os dados informados e gerar um ID (identificador único) para o cliente
- O Sistema deverá gravar os dados do cliente em arquivo.

## Classes

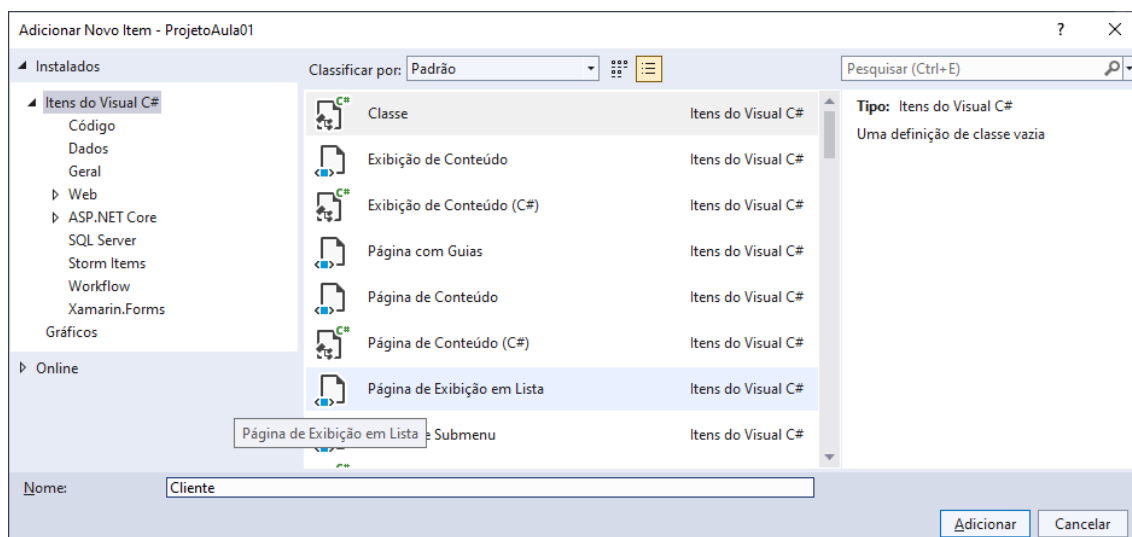
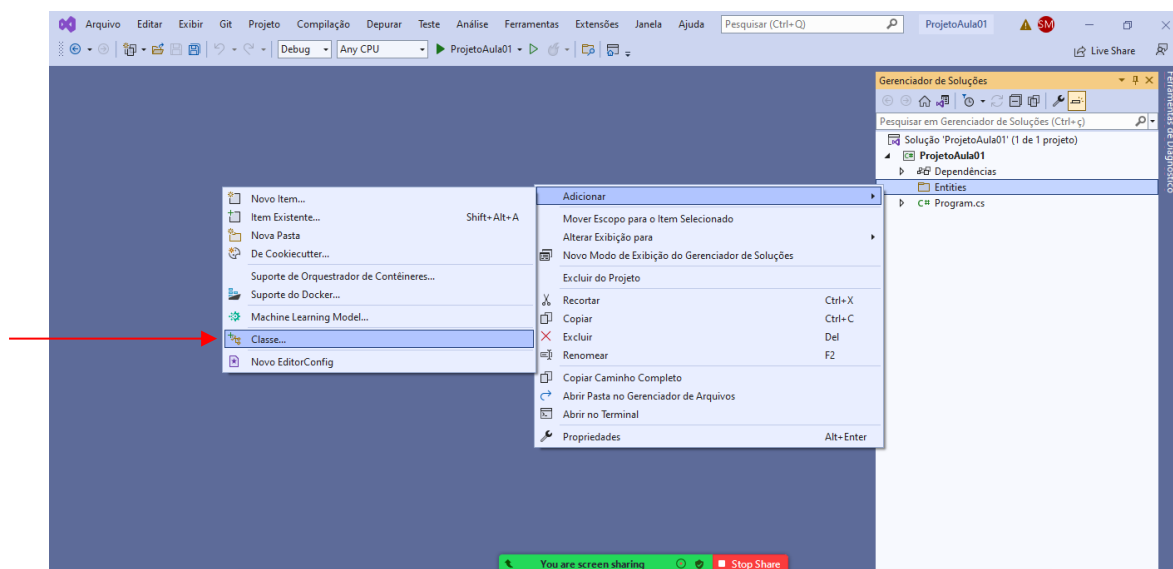
Estrutura de programação orientada a objetos composta de **atributos** (dados) e **métodos** (funções). Os atributos são os dados que declaramos em uma classe. Os métodos são as funções ou rotinas que criamos em uma classe. A primeira classe que vamos criar neste projeto será uma **entidade** para fazer a captura dos dados do cliente.

**/Entidades**



Em orientação a objetos, uma classe é uma descrição que abstrai um conjunto de objetos com características similares. Mais formalmente, é um conceito que encapsula abstrações de dados e procedimentos que descrevem o conteúdo e o comportamento de entidades do mundo real, representadas por objetos.

De outra forma, uma classe pode ser definida como uma descrição das propriedades ou estados possíveis de um conjunto de objetos, bem como os comportamentos ou ações aplicáveis a estes mesmos objetos. Linguagens de programação orientadas a objetos devem possibilitar a implementação de classes.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
//localização da classe no projeto
namespace ProjetoAula01.Entidades
{
    //definição da classe
    public class Cliente
    {
        //prop + 2x[tab]
        public Guid IdCliente { get; set; }
        public string Nome { get; set; }
        public string Email { get; set; }
        public string Telefone { get; set; }
        public DateTime DataCadastro { get; set; }
    }
}
```

## Modificadores de visibilidade:

### public

Modificador de visibilidade que permite acesso total a um elemento (classe atributo ou método).

### private

Modificador de visibilidade aplicado a atributos ou métodos. É o tipo mais restritivo de acesso, pois só permite acesso a um atributo ou método dentro da própria classe em que ele foi declarado.

---

Para cada atributo privado de uma classe podemos criar métodos públicos que permitam fazer o acesso aos atributos. Esses métodos são chamados de **set** e **get**.

## Encapsulamento

Prática para protegermos o conteúdo de uma classe do acesso externo, fazendo isso através de métodos de entrada e saída.

- **set** (operador para entrada de dados / atribuição)
- **get** (operador para saída de dados / retorno)

Encapsulamento vem de encapsular, que em programação orientada a objetos significa separar o programa em partes, o mais isoladas possível. A ideia é tornar o software mais flexível, fácil de modificar e de criar implementações.

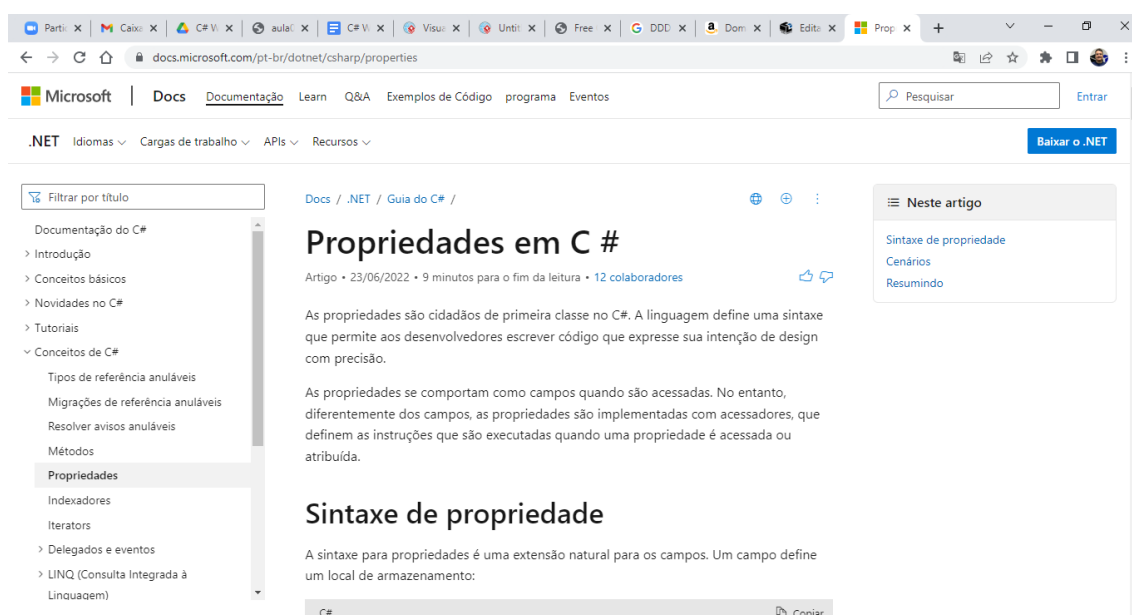
Para exemplificar, podemos pensar em uma dona de casa (usuário) utilizando um liquidificador (sistema). O usuário não necessita conhecer detalhes do funcionamento interno do sistema para poder utilizá-lo, precisa apenas conhecer a interface, no caso, os botões que controlam o liquidificador.

Uma grande vantagem do encapsulamento é que toda parte encapsulada pode ser modificada sem que os usuários da classe em questão sejam afetados. No exemplo do liquidificador, um técnico poderia substituir o

motor do equipamento por um outro totalmente diferente, sem que a dona de casa seja afetada - afinal, ela continuará somente tendo que pressionar o botão.

O encapsulamento protege o acesso direto (referência) aos atributos de uma instância fora da classe onde estes foram declarados. Esta proteção consiste em se usar modificadores de acesso mais restritivos sobre os atributos definidos na classe. Depois devem ser criados métodos para manipular de forma indireta os atributos da classe.

<https://docs.microsoft.com/pt-br/dotnet/csharp/properties>



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

//localização da classe no projeto
namespace ProjetoAula01.Entidades
{
    //definição da classe
    public class Cliente
    {
        //prop + 2x[tab]
        public Guid IdCliente { get; set; }
        public string Nome { get; set; }
        public string Email { get; set; }
        public string Telefone { get; set; }
        public DateTime DataCadastro { get; set; }
    }
}
```



## Objeto / variável de instância

Consiste em uma variável criada a partir da referência de uma classe.  
Também é chamado de variável de instância;

```
var cliente = new Cliente();  
           [Objeto]           [Instância da classe]  
           [Variável de instância]
```

```
using ProjetoAula01.Entidades;  
  
//Localização da classe no projeto  
namespace ProjetoAula01  
{  
    //declaração da classe  
    public class Program  
    {  
        //método (função) utilizada para executar o projeto  
        public static void Main(string[] args)  
        {  
            Console.WriteLine("\n *** CADASTRO DE CLIENTES *** \n");  
  
            //variável de instância / objeto  
            var cliente = new Cliente();  
  
            cliente.IdCliente = Guid.NewGuid();  
  
            Console.WriteLine("INFORME O NOME DO CLIENTE.....: ");  
            cliente.Nome = Console.ReadLine();  
  
            Console.WriteLine("INFORME O EMAIL DO CLIENTE....: ");  
            cliente.Email = Console.ReadLine();  
  
            Console.WriteLine("INFORME O TELEFONE DO CLIENTE.: ");  
            cliente.Telefone = Console.ReadLine();  
  
            cliente.DataCadastro = DateTime.Now;  
  
            //imprimindo os dados  
            Console.WriteLine("\nDADOS DO CLIENTE:");  
            Console.WriteLine($" \tID DO CLIENTE.....: {cliente.IdCliente}");  
            Console.WriteLine($" \tNOME DO CLIENTE....: {cliente.Nome}");  
            Console.WriteLine($" \tEMAIL.....: {cliente.Email}");  
            Console.WriteLine($" \tTELEFONE.....: {cliente.Telefone}");  
            Console.WriteLine($" \tDATA DE CADASTRO...:  
                                {cliente.DataCadastro}");  
  
            Console.ReadKey();  
        }  
    }  
}
```

```
*** CADASTRO DE CLIENTES ***

INFORME O NOME DO CLIENTE.....: Sergio Mendes
INFORME O EMAIL DO CLIENTE.....: sergio.coti@gmail.com
INFORME O TELEFONE DO CLIENTE.: 21 96957 5900

DADOS DO CLIENTE:
  ID DO CLIENTE.....: cee971b-6933-4ba5-99d8-e0f08b9777f6
  NOME DO CLIENTE.....: Sergio Mendes
  EMAIL.....: sergio.coti@gmail.com
  TELEFONE.....: 21 96957 5900
  DATA DE CADASTRO.....: 05/07/2022 23:21:12
\
```

Programação orientada a objetos (POO, ou OOP segundo as suas siglas em inglês) é um paradigma de programação baseado no conceito de "objetos", que podem conter dados na forma de campos, também conhecidos como atributos, e códigos, na forma de procedimentos, também conhecidos como métodos.

Uma característica de objetos é que um procedimento de objeto pode acessar, e geralmente modificar, os campos de dados do objeto com o qual eles estão associados

## SOLID (SRP, OCP, LSP, ISP, DIP)

Consiste em um acrônimo para 5 boas práticas de POO, são elas:

- **SRP** Princípio de responsabilidade única.
- **OCP** Princípio de aberto e fechado
- **ISP** Princípio de substituição de Liskov
- **ISP** Princípio de segmentação de interfaces
- **DIP** Princípio de inversão de dependência.



### SRP – Princípio de responsabilidade única.

Define que cada classe em um projeto deve ter uma única responsabilidade, mantendo a coesão de forma que os métodos de uma classe sejam voltados para resolver apenas 1 problema específico.



## SOLID - SRP

### Single Responsibility Principle

Na programação, o Princípio da responsabilidade única declara que cada módulo ou classe deve ter responsabilidade sobre uma única parte da funcionalidade fornecida pelo software.

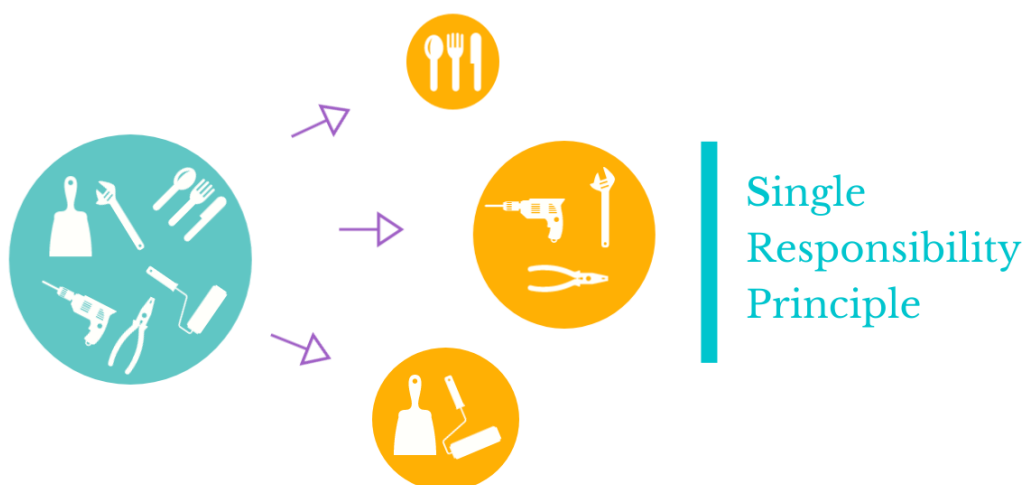
Você pode ter ouvido a citação: "Faça uma coisa e faça bem".

Isso se refere ao princípio da responsabilidade única.

No artigo citado acima, Robert C. Martin define uma responsabilidade como um "motivo para mudar" e conclui que uma classe ou módulo deve ter um e apenas um motivo para ser alterado.

Como esse princípio nos ajuda a criar um software melhor? Vamos ver alguns dos seus benefícios:

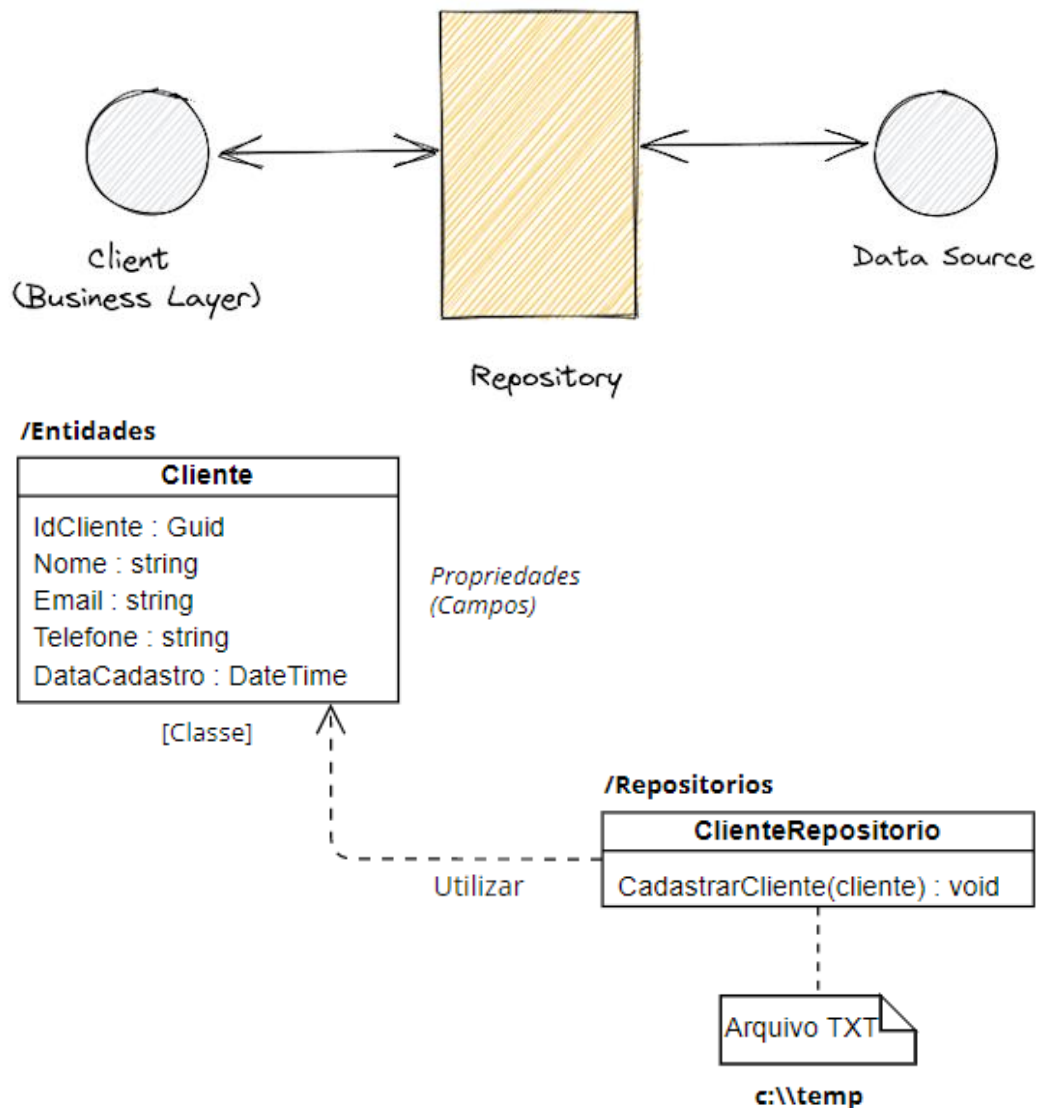
- **Teste** — Uma classe com uma responsabilidade terá muito menos casos de teste
- **Menor acoplamento** — menos funcionalidade em uma única classe terá menos dependências
- **Organização** — Classes menores e bem-organizadas são mais fáceis de pesquisar do que as classes monolíticas



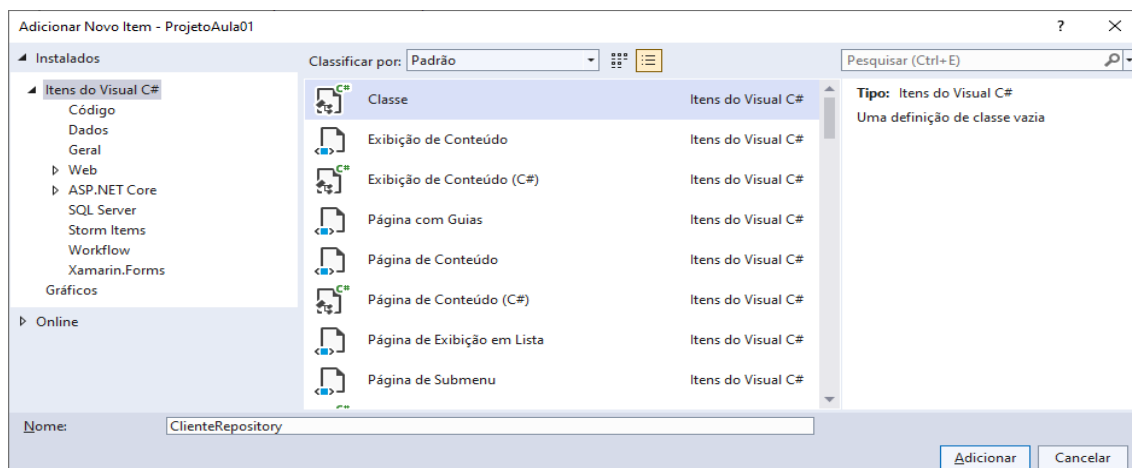
No nosso projeto iremos criar uma classe somente para fazer a exportação dos dados do cliente para arquivo. Essa classe terá o nome de

**ClienteRepositorio**

**\*\* Repositorio:** Nome dado para classes que fazem algum tipo de armazenamento de dados no projeto (arquivos ou banco de dados).



Criando a classe **ClienteRepositorio.cs**

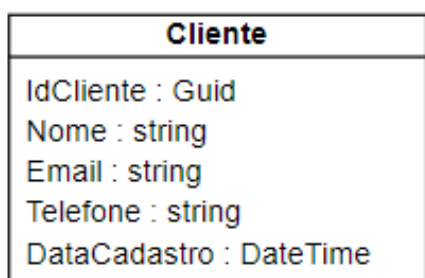


## void

Palavra reservada utilizada para métodos que não retornam valor.

Método do tipo "void" não retornam valor.

### /Entidades



Propriedades  
(Campos)

[Classe]

Utilizar

### /Repositorios

#### ClienteRepositorio

CadastrarCliente(cliente) : void

Arquivo TXT

c:\\temp

```
using ProjetoAula01.Entidades;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
```

```
//localização da classe no projeto
namespace ProjetoAula01.Repositorios
{
```

```
    //definição da classe
```

```
    public class ClienteRepositorio
    {
```

```
        //método para gravar os dados de um cliente em arquivo
```

```
        public void CadastrarCliente(Cliente cliente)
        {
```

```
            var diretorio = "c:\\temp";
```

```
            var arquivo = $"cliente_{cliente.IdCliente}.txt";
```

```
            //verificar se a pasta c:\\temp não existe
```

```
            if(!Directory.Exists(diretorio))
```

```
                Directory.CreateDirectory(diretorio); //criar a pasta
```

```
//abrindo um arquivo para gravação
var streamWriter = new StreamWriter($"{diretorio}\\{arquivo}");

//escrever os dados do arquivo
streamWriter.WriteLine($"ID DO CLIENTE...: {cliente.IdCliente}");
streamWriter.WriteLine($"NOME.....: {cliente.Nome}");
streamWriter.WriteLine($"EMAIL.....: {cliente.Email}");
streamWriter.WriteLine($"TELEFONE.....: {cliente.Telefone}");
streamWriter.WriteLine($"DATA DE CADASTRO:
                        {cliente.DataCadastro}");

//salvar e fechar o arquivo
streamWriter.Flush();
streamWriter.Close();
}
}
```

## Testando na classe Program:

Criando uma instância da classe ClienteRepository:

```
var clienteRepository = new ClienteRepository();
    [Variável de instância]           [Inicialização]
    (Objeto)
```

```
using ProjetoAula01.Entidades;
using ProjetoAula01.Repositorios;

//Localização da classe no projeto
namespace ProjetoAula01
{
    //declaração da classe
    public class Program
    {
        //método (função) utilizada para executar o projeto
        public static void Main(string[] args)
        {
            Console.WriteLine("\n *** CADASTRO DE CLIENTES *** \n");

            //variável de instância / objeto
            var cliente = new Cliente();

            cliente.IdCliente = Guid.NewGuid();

            Console.Write("INFORME O NOME DO CLIENTE.....: ");
            cliente.Nome = Console.ReadLine();

            Console.Write("INFORME O EMAIL DO CLIENTE.....: ");
            cliente.Email = Console.ReadLine();

            Console.Write("INFORME O TELEFONE DO CLIENTE.: ");
            cliente.Telefone = Console.ReadLine();

            cliente.DataCadastro = DateTime.Now;
```

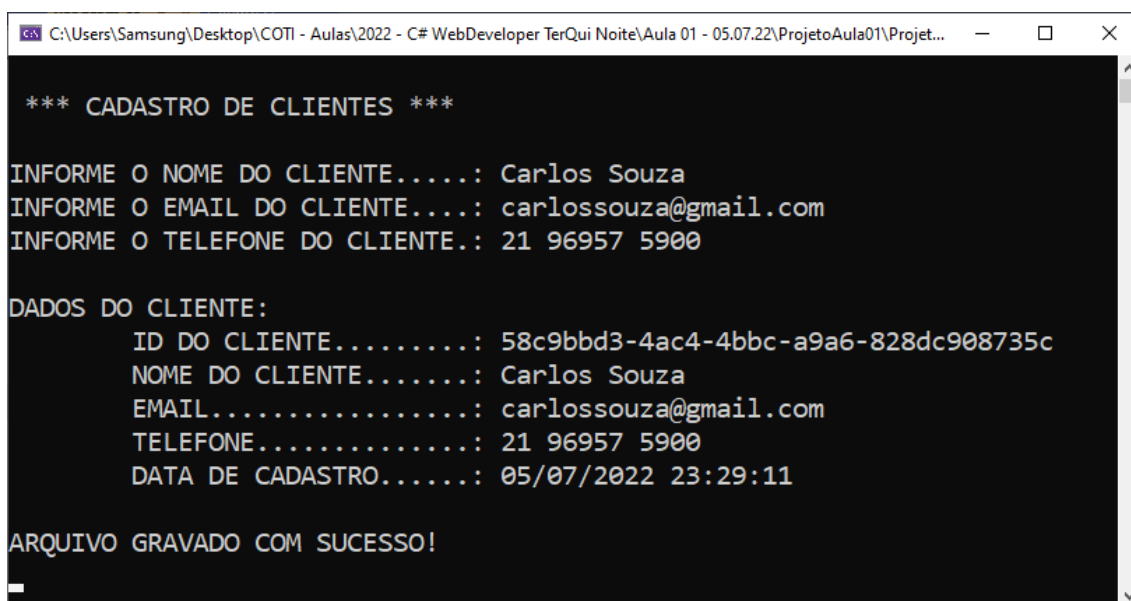
```
//imprimindo os dados
Console.WriteLine("\nDADOS DO CLIENTE:");
Console.WriteLine($" \tID DO CLIENTE.....: {cliente.IdCliente}");
Console.WriteLine($" \tNOME DO CLIENTE...: {cliente.Nome}");
Console.WriteLine($" \tEMAIL.....: {cliente.Email}");
Console.WriteLine($" \tTELEFONE.....: {cliente.Telefone}");
Console.WriteLine($" \tDATA DE CADASTRO...:
                        {cliente.DataCadastro}");

//variável de instância / objeto
var clienteRepositorio = new ClienteRepositorio();
//cadastrando o cliente
clienteRepositorio.CadastrarCliente(cliente);

Console.WriteLine("\nARQUIVO GRAVADO COM SUCESSO!");

Console.ReadKey();
}
}
```

## Resultado:



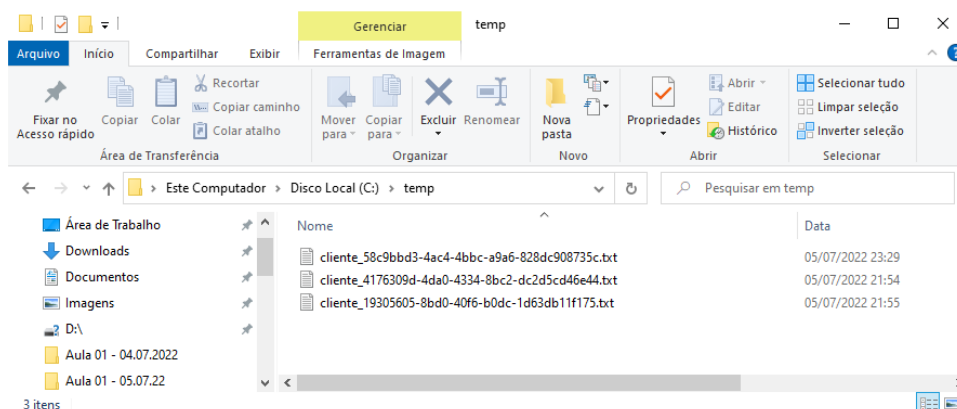
```
*** CADASTRO DE CLIENTES ***

INFORME O NOME DO CLIENTE.....: Carlos Souza
INFORME O EMAIL DO CLIENTE.....: carlossouza@gmail.com
INFORME O TELEFONE DO CLIENTE.: 21 96957 5900

DADOS DO CLIENTE:
ID DO CLIENTE.....: 58c9bbd3-4ac4-4bbc-a9a6-828dc908735c
NOME DO CLIENTE.....: Carlos Souza
EMAIL.....: carlossouza@gmail.com
TELEFONE.....: 21 96957 5900
DATA DE CADASTRO.....: 05/07/2022 23:29:11

ARQUIVO GRAVADO COM SUCESSO!
```

## Arquivos gerados:



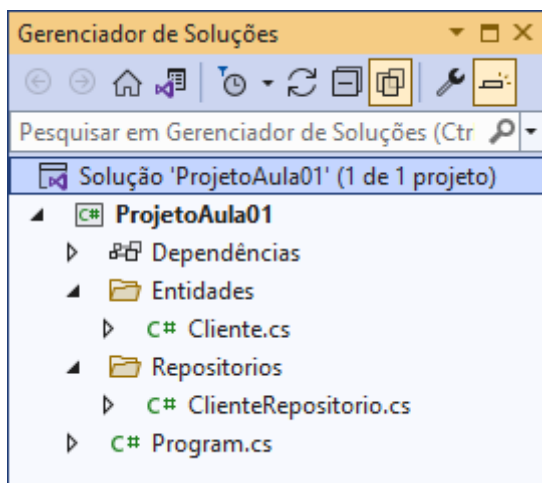


## Dados gravados:

```
cliente_58c9bbd3-4ac4-4bbc-a9a6-828dc908735c.txt - Bloco de Notas
Arquivo Editar Formatar Exibir Ajuda
ID DO CLIENTE.....: 58c9bbd3-4ac4-4bbc-a9a6-828dc908735c
NOME.....: Carlos Souza
EMAIL.....: carlossouza@gmail.com
TELEFONE.....: 21 96957 5900
DATA DE CADASTRO...: 05/07/2022 23:29:11
Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

```
cliente_4176309d-4da0-4334-8bc2-dc2d5cd46e44.txt - Bloco de Notas
Arquivo Editar Formatar Exibir Ajuda
ID DO CLIENTE.....: 4176309d-4da0-4334-8bc2-dc2d5cd46e44
NOME.....: Ana Paula
EMAIL.....: anapaula@gmail.com
TELEFONE.....: 21 96957 5900
DATA DE CADASTRO...: 05/07/2022 21:54:55
Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

## Estrutura do projeto:



## Classes

- As *classes* são os tipos do C# mais fundamentais. Uma classe é uma estrutura de dados que combina ações (métodos e outros membros da função) e estado (campos) em uma única unidade.
- Uma classe fornece uma definição para *instâncias* da classe criadas dinamicamente, também conhecidas como *objetos*.
- As classes dão suporte à *herança* e *polimorfismo*, mecanismos nos quais *classes derivadas* podem estender e especializar *classes base*.



## Modificadores de visibilidade:

---

### **public**

- Define acesso total para uma classe, atributo ou método.

### **internal**

- Permite acesso somente dentro do mesmo Assembly.

### **protected**

- Permite (para atributos ou métodos) acesso somente por meio de herança.

### **private**

- Permite (para atributos ou métodos) acesso somente dentro da própria classe onde o elemento foi declarado.

## Objeto

---

Consiste em uma variável criada a partir do espaço de memória de uma classe. Também é chamado de instancia da classe.

## Encapsulamento

---

Ao invés de declararmos os atributos como públicos, iremos mantê-los com visibilidade "private" e criar métodos que permitam acessar os atributos.

Um exemplo de encapsulamento ocorre quando uma classe declara seus atributos como privados e cria métodos públicos que permitem acessar indiretamente os atributos.

