

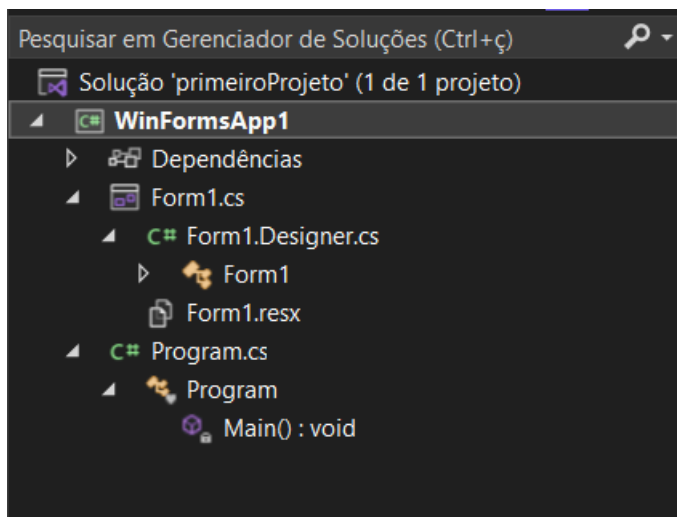
C# e Visual Studio

Criando projeto Solution:

Para a criação do projeto devemos criar uma **Solution**, uma Solution é um projeto que pode ter outros diversos projetos dentro dele: File → New Project → Installed → Templates → Other Project Types → Visual Studio Solutions → Blank Solution

Criando projeto Windows Forms:

Após a criação da Solution, podemos criar um projeto Windows Form dentro dessa solution clicando com o botão direito do mouse sobre a Solution → Adicionar → Novo Projeto → Windows Forms.



Podemos analisar que o projeto do tipo Windows Forms cria duas classes principais: **Form1.Designer.cs** e **Program.cs**.

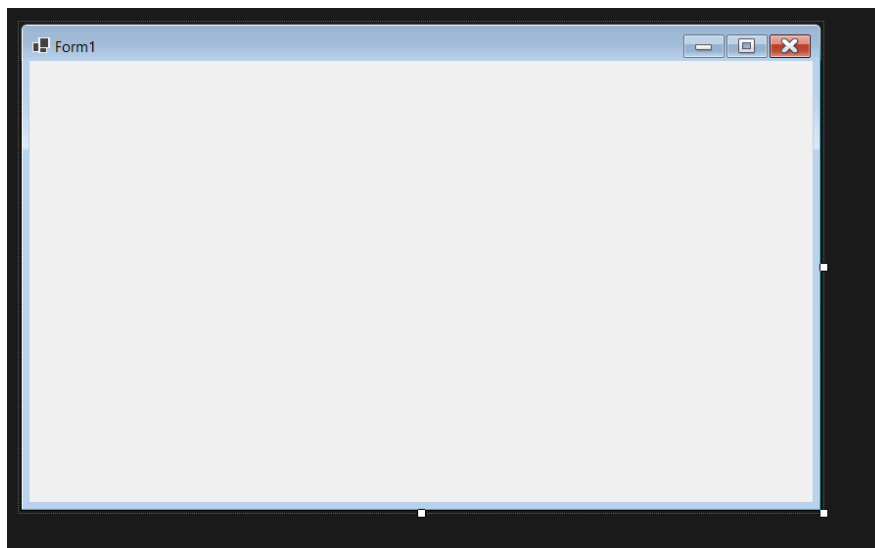
Form1.Designer.cs:

```
namespace WinFormsApp1
{
    3 referências
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        0 referências
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Windows Form Designer generated code
    }
}
```

Analisando a classe Form1 percebemos que ela é a classe responsável por criar o Form1:



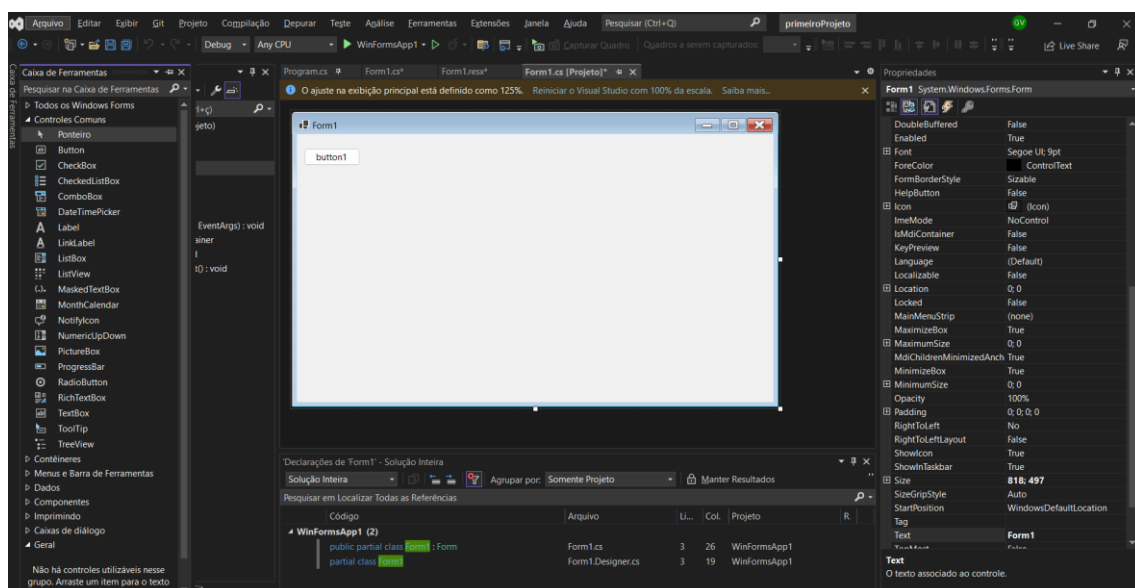
Program.cs:

```
namespace WinFormsApp1
{
    0 referências
    internal static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        0 referências
        static void Main()
        {
            // To customize application configuration such as set high DPI settings or default font,
            // see https://aka.ms/applicationconfiguration.
            ApplicationConfiguration.Initialize();
            Application.Run(new Form1());
        }
    }
}
```

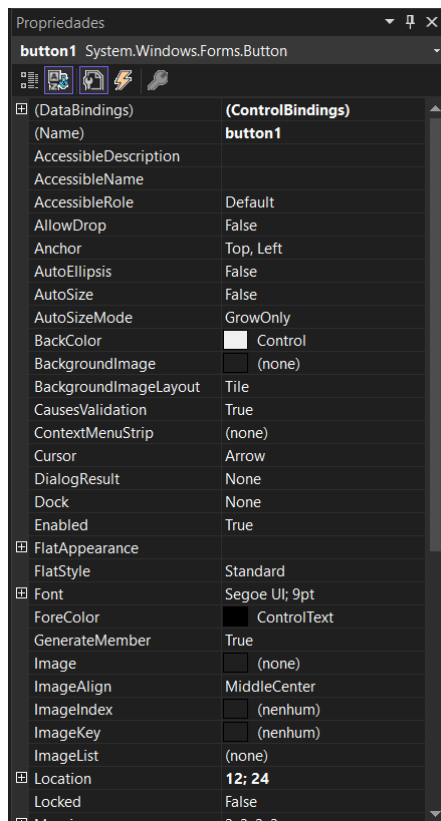
Analisando a classe Program, percebemos que ela é a classe responsável por conter e executar o nosso método main.

Caixa de ferramentas:

Quando falamos no Windows forms, possuímos uma coisa chamada Caixa de ferramentas que nos possibilita adicionar itens pre-prontos ao nosso projeto, sem ter que escrever o código do componente:



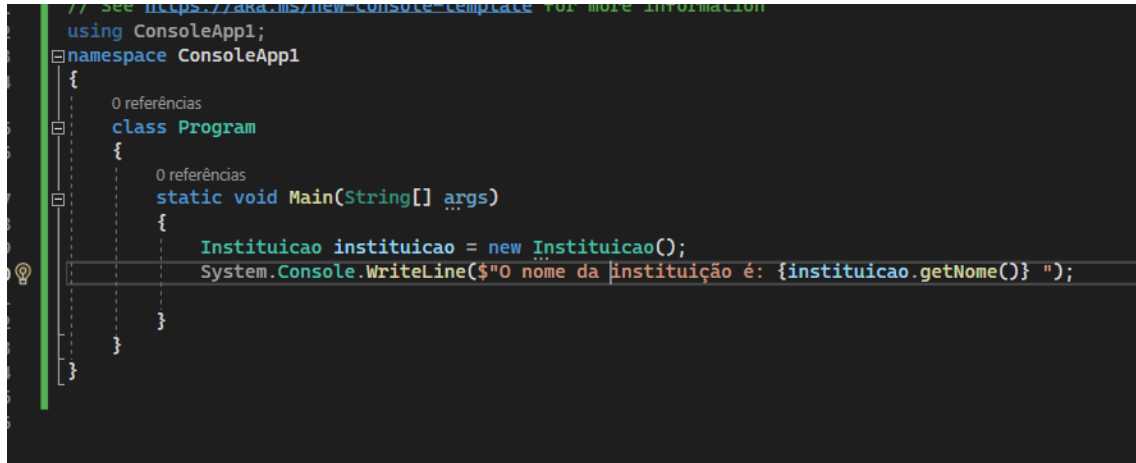
Cada item adicionado possuirá suas propriedades que poderão ser alteradas pela tabela de propriedades ou pelo próprio código:



```
private void InitializeComponent()
{
    button1 = new Button();
    SuspendLayout();
    //
    // button1
    //
    button1.Location = new Point(12, 24);
    button1.Name = "button1";
    button1.Size = new Size(94, 29);
    button1.TabIndex = 0;
    button1.Text = "button1";
    button1.UseVisualStyleBackColor = true;
    //
    // Form1
    //
    AutoScaleDimensions = new.SizeF(8F, 20F);
    AutoScaleMode = AutoScaleMode.Font;
    BackColor = SystemColors.Control;
    ClientSize = new Size(800, 450);
    Controls.Add(button1);
    Cursor = Cursors.Arrow;
    Name = "Form1";
    Text = "Form1";
    Load += Form1_Load;
    ResumeLayout(false);
}
```

Orientação a objetos:

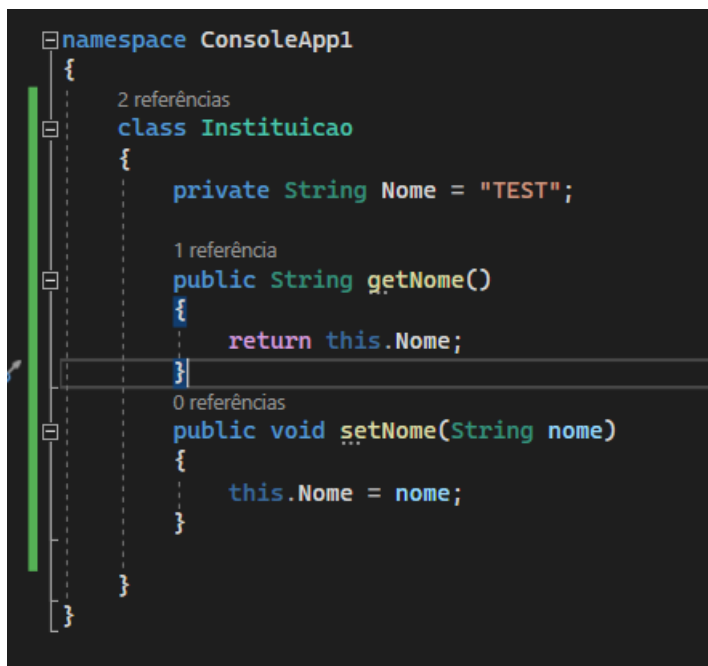
Quando falamos em orientação a objetos em C# estamos falando de algo muito parecido com o java, veja o exemplo de uma aplicação focada no console:



```
// See https://aka.ms/new-console-template for more information
using ConsoleApp1;

namespace ConsoleApp1
{
    0 referências
    class Program
    {
        0 referências
        static void Main(String[] args)
        {
            Instituicao instituicao = new Instituicao();
            System.Console.WriteLine($"O nome da instituição é: {instituicao.getNome()} ");
        }
    }
}
```

Essa é a classe “Main” da aplicação que só possui no momento um objeto de “Instituicao” e um método para imprimir no console o nome da instituição, agora se analisarmos a classe “Instituicao” veremos a seguinte estrutura:



```
namespace ConsoleApp1
{
    2 referências
    class Instituicao
    {
        private String Nome = "TEST";

        1 referência
        public String getNome()
        {
            return this.Nome;
        }

        0 referências
        public void setName(String nome)
        {
            this.Nome = nome;
        }
    }
}
```

Métodos da classe Console:

WriteLine → Usado para imprimir alguma coisa no console com quebra de linha.

Write → Usado para imprimir no console sem quebra de linha.

ReadLine → Usado para ler uma linha do console.

Read → Usado para ler apenas um Caractere digitado.

ReadKey → Usado para ler uma tecla clicada.

Melhorando a classe:

Perceba que anteriormente declaramos o método “get” e “set” porem isso não é necessário podemos simplesmente adicionar eles dessa forma:

```
namespace ConsoleApp1
{
    2 referências
    public class Aluno
    {
        1 referência
        public String NomeAluno { get; set; } = "Aluno";
        public Instituicao instituicao = new Instituicao();
    }
}
```

Herança:

Quando falamos de herança entre classes no C# estamos falando do operador “:” que define a herança entre classes:

```
namespace ConsoleApp1
{
    2 referências
    public class TesteHeranca : Instituicao
    {
    }
}
```

```
System.Console.WriteLine($"O nome da instituição é: {instituicao.getNome()} ");
System.Console.WriteLine($"teste Herança: {testeHeranca.getNome()}");
```

```
O nome da instituição é: TEST
teste Herança: TEST
```

É importante deixar claro que assim como o Java o C# não possibilita herança múltipla de classes apenas de interfaces.

Sobescrita de métodos:

Para se termos a sobescritas de métodos no c# eles devem estar marcados com os operadores “virtual” e “override”:

```
public class Animal
```

```
{
```

```

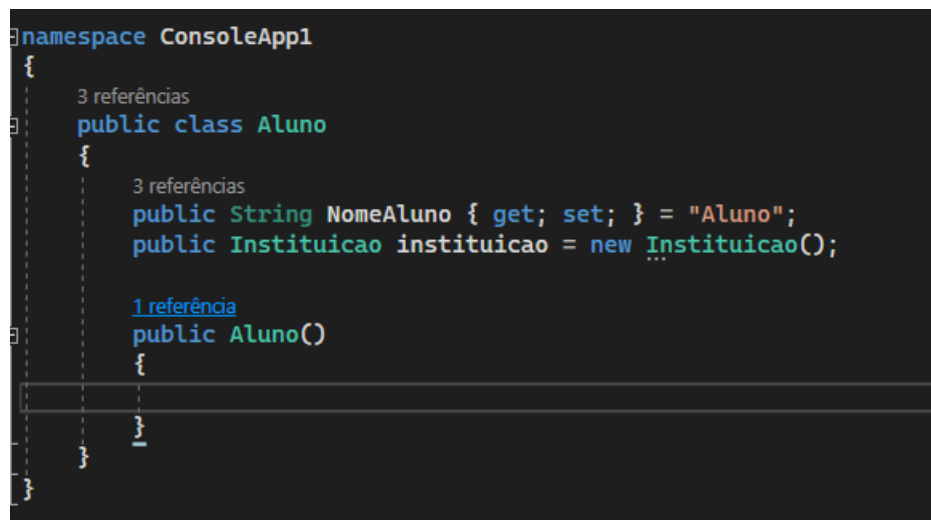
    public virtual void EmitSound()
    {
        Console.WriteLine("Animal emitindo som.");
    }
}

public class Cat : Animal
{
    public override void EmitSound()
    {
        Console.WriteLine("Miau!");
    }
}

```

Constructor:

Assim como nas demais linguagens o construtor da classe no C# é o método público com o nome da classe:



```

namespace ConsoleApp1
{
    3 referências
    public class Aluno
    {
        3 referências
        public String NomeAluno { get; set; } = "Aluno";
        public Instituicao instituicao = new Instituicao();

        1 referência
        public Aluno()
        {
        }
    }
}

```

Ao usar o constructor podemos alterar algumas sintaxes na inicialização da classe:

```
class Endereco
{
    public string Rua { get; set; }
    public string Numero { get; set; }
    public string Bairro { get; set; }
}

Endereco = new Endereco()
{
    Bairro = "Liberdade"
}
```

Coleções:

Quando falamos de arrays em c# eles podem ser collections ou arrays simples:

Arrays simples:

```
public Departamento[] Departamentos { get; } = new Departamento[10];
```

No array simples definimos um tipo pre-definido no array.

Collections:

```
public IList<Curso> Cursos { get; } = new List<Curso>()
```

No collections utilizamos o type IList que é um tipo genérico para as listas onde é passado o type das variáveis, em seguida inicializamos o objeto da Lista passando também o parâmetro da lista.

É importante analisar que o C# faz um método de repetição próprio para iterar sobre os arrays chamado de foreach():

```
foreach (var curso in dptoAlimentos.Cursos) { Console.WriteLine($"==> {curso.Nome}  
{curso.CargaHoraria}h"); }
```


Generics:

Os generics no C# funcionam igualmente no Java:

```
public void metodoTeste<T>(T a) { }
```

Reflection:

Para utilizarmos a reflexão em um objeto devemos primeiro importar as seguintes bibliotecas:

```
using System;  
using System.Reflection;
```

Através da biblioteca Reflection conseguimos ter os types e os métodos necessários exemplo:

```
class Program  
{  
    static void Main(string[] args)  
    {  
        // Obtém o tipo da classe MyClass  
        Type myType = typeof(MyClass);  
  
        // Obtém todas as propriedades públicas da classe  
        PropertyInfo[] properties = myType.GetProperties();  
  
        // Obtém todos os métodos públicos da classe  
        MethodInfo[] methods = myType.GetMethods();  
  
        // Imprime os nomes das propriedades na tela  
        Console.WriteLine("Properties:");  
        foreach (var property in properties)  
        {  
            Console.WriteLine(property.Name);  
        }  
  
        // Imprime os nomes dos métodos na tela  
        Console.WriteLine("\nMethods:");  
        foreach (var method in methods)  
        {  
            Console.WriteLine(method.Name);  
        }  
    }  
}
```

Tecnologias Unicas C#:

Propriedades:

As propriedades são métodos determinados métodos pre-prontos que se relacionam com as variáveis como os métodos get e set que são os mais comuns, veja o exemplo a seguir de todos os métodos possíveis nas propriedades:

```
using System;

public class ExemploPropriedades
{
    private int idade;
    private int valorInicial = 10;
    private EventHandler meuEvento;

    public int Idade
    {
        get
        {
            return idade;
        }
        set
        {
            idade = value;
        }
    }

    public int ValorInicial
    {
        init
        {
            valorInicial = value;
        }
    }

    public int ValorSomenteLeitura
    {
```

```

    get
    {
        return valorInicial;
    }
}

public event EventHandler MeuEvento
{
    add
    {
        meuEvento += value;
        Console.WriteLine("Assinando o evento.");
    }
    remove
    {
        meuEvento -= value;
        Console.WriteLine("Cancelando a assinatura do evento.");
    }
}

public int IdadeModificada
{
    set
    {
        idade = value;
        Console.WriteLine("Idade modificada para: " + idade);
        meuEvento?.Invoke(this, EventArgs.Empty);
    }
}
}

```

1. `public int Idade { get; set; }` Esta é uma propriedade automática, que define implicitamente um método get e um set para o campo `Idade`. Isso permite que o valor de `Idade` seja lido e alterado diretamente como uma propriedade.

2. `public int ValorInicial { init; }` Esta é uma propriedade somente de inicialização. Ela define um método `init`, que é usado para inicializar o valor da propriedade no momento da criação do objeto, mas não permite que o valor seja alterado posteriormente.
3. `public int ValorSomenteLeitura { get; }` Esta é uma propriedade somente leitura. Ela define apenas um método `get`, que permite que o valor da propriedade seja lido, mas não permite que seja alterado.
4. `public event EventHandler MeuEvento { add; remove; }` Esta é uma propriedade de evento, que define métodos `add` e `remove` para manipular eventos. Isso permite que os assinantes do evento sejam adicionados ou removidos dinamicamente.
5. `public int IdadeModificada { set; }` Esta é uma propriedade com um método `set` personalizado. Ele permite que o valor de `Idade` seja definido com um método `set` personalizado que executa uma ação adicional, neste caso, dispara um evento `MeuEvento` quando a idade é modificada.

Em resumo, cada propriedade neste código define um comportamento diferente para o campo correspondente, utilizando diferentes métodos (`get`, `set`, `init`, `add`, `remove`) para controlar o acesso e a modificação dos valores.

Delegados:

Delegados são tipos que permitem encapsular métodos com uma assinatura específica. Eles são muito úteis quando queremos passar um método como parâmetro para outro método ou quando queremos chamar um método em um objeto sem conhecer o método ou o objeto em tempo de compilação.

Em outras palavras, um delegado é um tipo que representa uma referência a um método com uma determinada assinatura. Podemos usar um delegado para declarar uma variável que pode apontar para qualquer método com a mesma assinatura. Quando chamamos um delegado, ele chama o método que está referenciado.

Aqui está um exemplo de como declarar e usar um delegado em C#:

```
delegate int OperacaoMatematica(int x, int y);
```

```
// Método que recebe um delegado como parâmetro
```

```
static void RealizarOperacao(OperacaoMatematica operacao, int x, int y)
```

```
{
```

```
int resultado = operacao(x, y);

Console.WriteLine("Resultado da operação: " + resultado);
}
```

Em outras palavras Delegados são uma forma orientada a objetos de passar um método para uma função igual o js faz com os call-backs, porem aqui nos armazenamos o método em uma variável.

Eventos:

Eventos são uma forma de comunicação entre objetos em C#. Eles permitem que um objeto notifique outros objetos quando algo importante acontece, como uma mudança de estado. Quando um evento é disparado, todos os objetos que se inscreveram para receber notificações desse evento são notificados e podem tomar alguma ação.

Em C#, eventos são implementados usando a palavra-chave event. Para declarar um evento, precisamos primeiro definir o tipo de delegado que será usado para representar o evento. O delegado especifica a assinatura do método que será chamado quando o evento for disparado. Em seguida, podemos declarar uma variável do tipo delegado e marcá-la com a palavra-chave event.

Por exemplo, vamos supor que temos uma classe Contador que possui um evento ValorAlterado que é disparado toda vez que o valor do contador é alterado:

```
public class Contador
{
    private int valor;

    // Delegado que representa o evento
    public delegate void ValorAlteradoEventHandler(object sender, EventArgs e);

    // Evento que é disparado quando o valor do contador é alterado
    public event ValorAlteradoEventHandler ValorAlterado;

    public int Valor
    {
        get { return valor; }
        set
```

```

    {
        valor = value;
        // Dispara o evento ValorAlterado
        OnValorAlterado();
    }
}

protected virtual void OnValorAlterado()
{
    ValorAlterado?.Invoke(this, EventArgs.Empty);
}
}

```

LINQ:

LINQ (Language Integrated Query) é uma tecnologia da plataforma .NET que permite a consulta de dados em diferentes tipos de fontes de dados, como bancos de dados, coleções de objetos, arquivos XML, entre outros.

Exemplo:

```

List<Pessoa> pessoas = new List<Pessoa> {
    new Pessoa { Nome = "João", Idade = 30 },
    new Pessoa { Nome = "Maria", Idade = 25 },
    new Pessoa { Nome = "Pedro", Idade = 35 },
};

```

```

var pessoasComMaisDe30Anos = from p in pessoas
    where p.Idade > 30
    select p;

```

```

foreach (var pessoa in pessoasComMaisDe30Anos) {
    Console.WriteLine(pessoa.Nome);
}

```

A documentação pode ser acessada aqui: <https://learn.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries?redirectedfrom=MSDN>

Classes abstratas:

Assim como o Ts o C# também possui suporte a classes abstratas que não podem ser instanciadas em objetos apenas podem ser usadas pela herança:

```
public abstract class MinhaClasseAbstrata
{
    public abstract void MeuMetodoAbstrato();
}

public class MinhaClasseFilha : MinhaClasseAbstrata
{
    public override void MeuMetodoAbstrato()
    {
        // Implementação do método abstrato aqui
    }
}
```

Exceções:

No C# as exceções são tratadas através do bloco **try-catch** e todas as exceções podem ser declaradas pelo type Exception.

Para se Criar uma exceção personalizada devemos criar uma classe que estende a classe Exception:

```
using System;
```

```
public class IdadeInvalidaException : Exception
{
    public IdadeInvalidaException(int idade)
        : base($"Idade inválida: {idade}")
    {
        if (idade < 0 || idade > 100)
        {
            throw new ArgumentException("A idade fornecida é inválida.", nameof(idade));
        }
    }
}
```

