Programação Orienta à Objetos II - Revisão💻

👨🏾**Professor:** Salatiel Luz Marinho

📧salatiel.marinho@docente.unip.br

Conceitos sobre programação orientada à objetos

Em programação orientada a objetos (POO), uma classe é uma estrutura de dados que combina estado (atributos ou propriedades) e comportamentos (métodos ou funções). Ela é uma espécie de "blueprint" ou modelo a partir do qual os objetos são criados.

Em C#, por exemplo, você pode definir uma classe "Carro" com atributos como "Marca", "Modelo", "Cor" e métodos como "Acelerar" e "Frear". Cada objeto criado a partir desta classe, ou seja, cada instância de "Carro", terá esses atributos e métodos.

Aqui está um exemplo simples de uma classe em C#:

```
public class Carro
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
    public string Cor { get; set; }

    public void Acelerar()
    {
        Console.WriteLine("O carro está acelerando.");
    }
```

```
    public void Frear()
    {
        Console.WriteLine("O carro está freiando.");
    }
}
```

A classe é um dos principais pilares da POO, juntamente com a herança, o encapsulamento e o polimorfismo.

Em Programação Orientada a Objetos (POO), um método é uma operação que um objeto pode realizar. É uma sub-rotina associada a uma classe específica. Métodos são usados para expressar comportamentos dos objetos e geralmente operam em dados que são contidos nos atributos do objeto.

No contexto do C#, um método consiste em um bloco de código que executa uma ação específica. Ele pode aceitar valores de entrada, chamados parâmetros, e pode retornar um valor de saída. A definição de um método inclui a declaração de seu nome, o tipo de valor que ele retorna (ou void se não retornar nada), e a lista de parâmetros.

Por exemplo, na classe 'Carro' descrita no documento, os métodos 'Acelerar' e 'Frear' são operações que um objeto do tipo 'Carro' pode realizar. Quando chamamos `carro.Acelerar()`, estamos dizendo que o carro (um objeto da classe 'Carro') está realizando a ação de acelerar.

Um ponto importante sobre os métodos em C# é que eles podem ser sobrecarregados. A sobrecarga de métodos é um conceito importante na programação orientada a objetos. Ela permite que uma classe tenha vários métodos com o mesmo nome, mas com parâmetros diferentes.

Por exemplo, você pode ter um método chamado `Somar` em uma classe `Calculadora` que execute a soma de dois inteiros e outro método também chamado `Somar` que execute a soma de dois números de ponto flutuante (doubles). O compilador de C# escolhe automaticamente o método correto com base nos argumentos passados.

Veja um exemplo prático de sobrecarga de métodos em C#:

```
class Calculadora
{
    public int Somar(int a, int b)
```

```
    {
        return a + b;
    }

    public double Somar(double a, double b)
    {
        return a + b;
    }
}

class Program
{
    static void Main()
    {
        Calculadora calc = new Calculadora();
        int resultadoInt = calc.Somar(5, 3); // Chama o mét
odo com inteiros
        double resultadoDouble = calc.Somar(2.5, 1.5); // C
hama o método com doubles

        Console.WriteLine($"Resultado (int): {resultadoIn
t}");
        Console.WriteLine($"Resultado (double): {resultadoD
ouble}");
    }
}
```

Neste exemplo, a classe `Calculadora` possui dois métodos chamados `Somar`. Um aceita dois inteiros e retorna um inteiro, enquanto o outro aceita dois doubles e retorna um double. O compilador decide qual método chamar com base nos argumentos passados.

A sobrecarga de métodos traz benefícios como clareza, pois permite usar o mesmo nome para operações semelhantes, flexibilidade, pois é adaptável a diferentes tipos de dados, e reutilização de código, evitando duplicação de lógica.

Em C#, os tipos de variáveis são categorizados em dois tipos principais: tipos de valor e tipos de referência.

Tipos de valor são armazenados diretamente na memória da pilha e incluem:

- Tipos numéricos integrais: `byte`, `sbyte`, `int`, `uint`, `short`, `ushort`, `long`, `ulong`
- Tipos numéricos em ponto flutuante: `float`, `double`
- Tipo numérico decimal: `decimal`
- Tipo booleano: `bool`
- Tipo caractere: `char`
- Tipos de enumeração: `enum`
- Tipos de estrutura: `struct`

Tipos de referência são armazenados no heap e a variável na pilha contém uma referência ao espaço de memória no heap. Os tipos de referência incluem:

- Tipo de objeto: `object`
- Tipo de string: `string`
- Tipos de classe: `class`
- Tipos de interface: `interface`
- Tipos de array: `array`
- Tipo de delegado: `delegate`

Além disso, C# também suporta tipos anuláveis que permitem que tipos de valor armazenem `null`.

Aqui está um exemplo simples de como declarar diferentes tipos de variáveis em C#:

```
int numeroInteiro = 10;
double numeroDouble = 20.5;
bool valorBooleano = true;
char caracter = 'A';
string texto = "Hello, World!";
int? numeroNulo = null;
```

Os intervalos dos tipos inteiros em C# são os seguintes:

- `byte`: de 0 a 255
- `sbyte`: de -128 a 127
- `int`: de -2,147,483,648 a 2,147,483,647
- `uint`: de 0 a 4,294,967,295
- `short`: de -32,768 a 32,767
- `ushort`: de 0 a 65,535
- `long`: de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
- `ulong`: de 0 a 18,446,744,073,709,551,615

Herança: A herança é um conceito fundamental na programação orientada a objetos, e em C#, ela permite que uma classe (classe derivada) herde campos, propriedades e métodos de outra classe (classe base). Aqui estão alguns exemplos simples de herança em C#:

```
// Classe base Animal
public class Animal
{
    public string Nome { get; set; }
    public int Idade { get; set; }

    public void Comer()
    {
        Console.WriteLine($"{Nome} está comendo.");
    }

    public void Dormir()
    {
        Console.WriteLine($"{Nome} está dormindo.");
    }
}

// Classe derivada Gato que herda de Animal
```

```
public class Gato : Animal
{
    public string Raca { get; set; }

    public void Miar()
    {
        Console.WriteLine($"{Nome} está miando.");
    }
}

// Classe derivada Cachorro que herda de Animal
public class Cachorro : Animal
{
    public string CorDoPelo { get; set; }

    public void Latir()
    {
        Console.WriteLine($"{Nome} está latindo.");
    }
}
```

No exemplo acima, `Gato` e `Cachorro` são classes derivadas que herdam da classe base `Animal`. Elas podem usar os métodos `Comer()` e `Dormir()` da classe base e também têm seus próprios métodos, `Miar()` e `Latir()`, respectivamente.

Para usar essas classes em um programa:

```
class Program
{
    static void Main(string[] args)
    {
        Gato gato = new Gato { Nome = "Mingau", Idade = 3, Ra
        gato.Comer(); // Chamada do método herdado de Animal
        gato.Miar();  // Chamada do método específico de Gato

        Cachorro cachorro = new Cachorro { Nome = "Rex", Idad
```

```
        cachorro.Comer(); // Chamada do método herdado de Ani
        cachorro.Latir(); // Chamada do método específico de
    }
}
```

Este é um exemplo básico para ilustrar como a herança funciona em C#. Você pode expandir essas classes com mais propriedades e métodos conforme necessário para o seu projeto.

Encapsulamento é um conceito fundamental na **programação orientada a objetos (POO)**, e é usado para organizar e proteger os detalhes internos de uma classe. Vamos explorar o que é encapsulamento em C# e como aplicá-lo:

1. **Definição de Encapsulamento:**
    - O encapsulamento envolve **agrupar dados (atributos)** e **métodos** relacionados em uma classe.
    - Ele permite que os detalhes internos da implementação da classe permaneçam **ocultos** para os objetos que a utilizam.
    - A ideia é **separar o programa em partes isoladas**, tornando o software mais flexível e fácil de modificar.
2. **Modificadores de Acesso:**
    - **Públicos (public)**: São acessíveis externamente e fazem parte da interface pública da classe.
    - **Privados (private)**: São acessíveis apenas dentro da própria classe e não devem ser expostos externamente.
3. **Benefícios do Encapsulamento:**
    - **Ocultação de Detalhes**: Os usuários dos objetos não precisam conhecer a implementação interna.
    - **Modificação Interna**: Podemos alterar um objeto internamente sem afetar outras partes do sistema.
    - **Desenvolvimento Simplificado**: Os usuários não precisam saber como os objetos são construídos internamente.

- **Flexibilidade**: A implementação de um comportamento pode ser modificada sem impactar o restante do programa.

4. **Exemplo Prático em C#:**
   - Vamos criar uma classe chamada `Produto` com atributos privados (como preço e quantidade em estoque).
   - Usaremos métodos públicos para atualizar esses atributos e obter informações sobre o produto.

```
using System;
using System.Collections.Generic;

namespace SistemaEstoqueAgricola
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crie uma lista para armazenar os produtos
            List<Produto> produtos = new List<Produto>();

            // Exemplo: Adicione alguns produtos iniciais
            produtos.Add(new Produto("Arroz", 10.5, 100));
            produtos.Add(new Produto("Feijão", 8.0, 80));

            // Exemplo: Mostrar os produtos e seus estoques
            Console.WriteLine("Produtos no estoque:");
            foreach (var produto in produtos)
            {
                Console.WriteLine($"{produto.Nome} -
                 Preço: {produto.Preco:C} - Estoque: {produto
            }

            // Exemplo: Atualizar o estoque de um produto
            Console.Write("Digite o nome do produto para atua
            string nomeProduto = Console.ReadLine();
            Console.Write("Digite a quantidade a ser adiciona
```

```
             ou removida (negativa): ");
            int quantidadeAlteracao = int.Parse(Console.ReadL

            Produto produtoSelecionado =
            produtos.Find(p =>
            p.Nome.Equals(nomeProduto, StringComparison.Ordin
            if (produtoSelecionado != null)
            {
                produtoSelecionado.AtualizarEstoque(quantidad
                Console.WriteLine($"Estoque atualizado para
                {produtoSelecionado.QuantidadeEmEstoque} unid
            }
            else
            {
                Console.WriteLine("Produto não encontrado.");
            }
        }
    }

    class Produto
    {
        public string Nome { get; }
        public double Preco { get; }
        public int QuantidadeEmEstoque { get; private set; }

        public Produto(string nome, double preco, int quantid
        {
            Nome = nome;
            Preco = preco;
            QuantidadeEmEstoque = quantidadeInicial;
        }

        public void AtualizarEstoque(int quantidade)
        {
            QuantidadeEmEstoque += quantidade;
        }
    }
}
```

A **sobrecarga de métodos** é um conceito importante na **programação orientada a objetos (POO)**. Ela permite que uma classe tenha vários métodos com o **mesmo nome**, mas com **parâmetros diferentes**. Vamos explorar como funciona a sobrecarga de métodos em C#:

1. **Definição:**
    - A sobrecarga de métodos permite que você defina **vários métodos com o mesmo nome** em uma classe.
    - Esses métodos devem ter **assinaturas diferentes**, ou seja, tipos ou números diferentes de parâmetros.
    - O compilador escolhe automaticamente o método correto com base nos argumentos passados.
2. **Exemplo Prático:**

```
class Calculadora
{
    public int Somar(int a, int b)
    {
        return a + b;
    }

    public double Somar(double a, double b)
    {
        return a + b;
    }
}

class Program
{
    static void Main()
    {
        Calculadora calc = new Calculadora();
        int resultadoInt = calc.Somar(5, 3); // Chama o método
        double resultadoDouble = calc.Somar(2.5, 1.5); // Cha
```

```
        Console.WriteLine($"Resultado (int): {resultadoInt}")
        Console.WriteLine($"Resultado (double): {resultadoDou
    }
}
```

Neste exemplo, a classe `Calculadora` possui dois métodos chamados `Somar`.Um aceita dois inteiros e retorna um inteiro, enquanto o outro aceita dois doubles e retorna um double.O compilador decide qual método chamar com base nos argumentos passados.

Vantagens da Sobrecarga:

- **Clareza**: Permite usar o mesmo nome para operações semelhantes.
- **Flexibilidade**: Adaptável a diferentes tipos de dados.
- **Reutilização de Código**: Evita duplicação de lógica.

Propriedades Get e Set em C#:

Em C#, as propriedades `get` e `set` são usadas para obter e definir os valores de uma propriedade de um objeto, respectivamente. Elas permitem que você controle o acesso a um campo de uma classe, funcionando como uma espécie de "ponte" entre o campo e o código externo que deseja acessar esse campo.

As propriedades `get` e `set` em C# são muito úteis para implementar o princípio do encapsulamento, um dos pilares da programação orientada a objetos.

Aqui temos um exemplo simples:

```
public class Pessoa
{
    private string nome;

    public string Nome
    {
        get { return nome; }
        set { nome = value; }
```

```
    }
}
```

Neste exemplo, `Nome` é uma propriedade da classe `Pessoa`. O método `get` retorna o valor do campo `nome`, enquanto o método `set` modifica o valor do campo `nome`. A palavra-chave `value` é usada no `set` para representar o valor que está sendo atribuído à propriedade.

Essas propriedades podem ser usadas da seguinte maneira:

```
Pessoa pessoa = new Pessoa();
pessoa.Nome = "João";  // A propriedade 'set' é usada aq
ui.
Console.WriteLine(pessoa.Nome);  // A propriedade 'get'
é usada aqui.
```

Este exemplo imprimirá "João" na console.

Exemplo: Sistema de Gestão Agrícola

Para ilustrar a aplicação dos conceitos de orientação a objetos em um contexto real, podemos pensar em um sistema de gestão agrícola. Vamos supor que temos diferentes tipos de plantações, cada uma com suas características e métodos específicos.

Primeiro, criamos uma classe base chamada `Plantação` com propriedades e métodos comuns a todas as plantações.

```
public class Plantacao
{
    public string Nome { get; set; }
    public int Quantidade { get; set; }

    public void Plantar()
    {
        Console.WriteLine($"{Nome} está sendo plantad
o.");
    }

    public void Colher()
```

```
    {
        Console.WriteLine($"{Nome} está sendo colhid
o.");
    }
}
```

Em seguida, criamos classes derivadas para tipos específicos de plantações, como `Milho` e `Soja`, que herdam da classe base `Plantacao`. Cada uma dessas classes pode ter propriedades e métodos adicionais.

```
public class Milho : Plantacao
{
    public void Fertilizar()
    {
        Console.WriteLine("O milho está sendo fertilizad
o.");
    }
}

public class Soja : Plantacao
{
    public void Irrigar()
    {
        Console.WriteLine("A soja está sendo irrigad
a.");
    }
}
```

Agora podemos instanciar objetos dessas classes e usar seus métodos.

```
Milho milho = new Milho { Nome = "Milho", Quantidade = 2
00 };
milho.Plantar();
milho.Fertilizar();

Soja soja = new Soja { Nome = "Soja", Quantidade = 300
};
```

```
soja.Plantar();
soja.Irrigar();
```

Este exemplo demonstra como a programação orientada a objetos em C# pode ser usada para modelar situações do mundo real de maneira intuitiva e reutilizável.

Aqui está um exemplo de como usar `String.Format` em C#:

```
string nome = "João";
int idade = 30;

string mensagem = String.Format("Olá, meu nome é {0} e e
u tenho {1} anos.", nome, idade);

Console.WriteLine(mensagem);  // Isto imprimirá: "Olá, m
eu nome é João e eu tenho 30 anos."
```

Neste exemplo, `{0}` e `{1}` são substituídos pelos valores das variáveis `nome` e `idade`, respectivamente. `String.Format` é útil para criar strings que incluem valores de variáveis ou expressões.

String.IsNullOrEmpty em C#

O método `String.IsNullOrEmpty` em C# é usado para verificar se uma string especificada é `null` ou uma string vazia. Ele retorna `true` se a string é `null` ou se ela é uma string vazia (ou seja, tem zero caracteres), e `false` se a string contém um ou mais caracteres.

Aqui está um exemplo de como você pode usar `String.IsNullOrEmpty` em C#:

```
string str1 = null;
string str2 = "";
string str3 = "Olá, Mundo!";

Console.WriteLine(String.IsNullOrEmpty(str1));  // Isto
imprimirá: 'True'
Console.WriteLine(String.IsNullOrEmpty(str2));  // Isto
```

```
imprimirá: 'True'
Console.WriteLine(String.IsNullOrEmpty(str3));  // Isto
imprimirá: 'False'
```

Neste exemplo, `String.IsNullOrEmpty` retorna `true` para `str1` e `str2`, porque `str1` é `null` e `str2` é uma string vazia. Para `str3`, que contém a string "Olá, Mundo!", `String.IsNullOrEmpty` retorna `false`.

Este método é útil quando você quer verificar se uma string é `null` ou vazia antes de realizar operações nela, o que pode ajudar a prevenir erros em seu código.

If/Else em C#

O controle de fluxo `if/else` é uma estrutura de decisão básica em C#.

O comando `if` avalia uma expressão e, se essa expressão for `true`, o bloco de código dentro do `if` é executado. Se for `false`, o bloco de código é ignorado.

Aqui está um exemplo simples:

```
int numero = 10;

if (numero > 5)
{
    Console.WriteLine("O número é maior que 5.");
}
```

Neste caso, como o número 10 é maior que 5, a frase "O número é maior que 5." será impressa na console.

Você também pode adicionar um comando `else` após o `if`. O bloco de código dentro do `else` é executado se a expressão no `if` for `false`.

```
int numero = 4;

if (numero > 5)
{
    Console.WriteLine("O número é maior que 5.");
}
```

```
else
{
    Console.WriteLine("O número é menor ou igual a 5.");
}
```

Neste exemplo, como o número 4 não é maior que 5, a frase "O número é menor ou igual a 5." será impressa na console.

Além disso, você pode usar `else if` para adicionar condições adicionais. Se a expressão no `if` for `false`, as expressões no `else if` serão avaliadas na ordem em que aparecem. Quando uma expressão `true` é encontrada, o bloco de código correspondente é executado. Se todas as expressões forem `false`, o bloco de código no `else` (se houver) é executado.

```
int numero = 5;

if (numero > 5)
{
    Console.WriteLine("O número é maior que 5.");
}
else if (numero < 5)
{
    Console.WriteLine("O número é menor que 5.");
}
else
{
    Console.WriteLine("O número é igual a 5.");
}
```

Neste exemplo, como o número 5 não é nem maior nem menor que 5, a frase "O número é igual a 5." será impressa na console.

Switch/Case em C#

O comando `switch/case` em C# é uma estrutura de controle de fluxo que permite a execução de diferentes blocos de código com base no valor de uma expressão. É uma alternativa ao uso de várias instruções `if/else`.

Aqui está um exemplo simples de como usar `switch/case` em C#:

```
int diaDaSemana = 3;

switch (diaDaSemana)
{
    case 1:
        Console.WriteLine("Segunda-feira");
        break;
    case 2:
        Console.WriteLine("Terça-feira");
        break;
    case 3:
        Console.WriteLine("Quarta-feira");
        break;
    case 4:
        Console.WriteLine("Quinta-feira");
        break;
    case 5:
        Console.WriteLine("Sexta-feira");
        break;
    case 6:
        Console.WriteLine("Sábado");
        break;
    case 7:
        Console.WriteLine("Domingo");
        break;
    default:
        Console.WriteLine("Valor inválido");
        break;
}
```

Neste exemplo, a expressão `diaDaSemana` é avaliada e o bloco de código correspondente ao valor dessa expressão é executado. Se `diaDaSemana` for 1, a frase "Segunda-feira" será impressa na console. Se `diaDaSemana` for 2, a frase "Terça-feira" será impressa, e assim por diante. Se `diaDaSemana` não for nenhum dos valores especificados nos casos (1 a 7), o bloco de código no `default` é executado e a frase "Valor inválido" será impressa.

Note o uso da palavra-chave `break` após cada bloco de código em um caso. Isso é necessário para encerrar a execução do bloco de código e sair do `switch`. Se você esquecer de incluir `break`, o código continuará a executar o próximo caso, mesmo que a expressão não corresponda a ele, o que pode levar a resultados inesperados.

O comando `switch/case` em C# é uma maneira eficiente e legível de lidar com múltiplas condições baseadas em um único valor ou expressão.

Aqui está um exemplo simples de como bloquear a digitação de letras usando `Console.ReadLine()` em C#:

```
string entrada;
int numero;

Console.Write("Digite um número: ");
entrada = Console.ReadLine();

bool ehNumero = int.TryParse(entrada, out numero);

while (!ehNumero)
{
    Console.Write("Entrada inválida. Por favor, digite u
m número: ");
    entrada = Console.ReadLine();
    ehNumero = int.TryParse(entrada, out numero);
}

Console.WriteLine("Você digitou o número " + numero);
```

Nesse exemplo, o método `int.TryParse()` tenta converter a string de entrada para um número inteiro. Se a conversão for bem-sucedida, ele retorna `true`, caso contrário, retorna `false`. Se o usuário digitar letras, a conversão falhará, o loop continuará e o usuário será solicitado a digitar novamente até que um número seja inserido.

Aqui está um exemplo de como usar a classe `DateTime` para extrair o dia, mês e ano em C#:

```
DateTime dataAtual = DateTime.Now;

int dia = dataAtual.Day;
int mes = dataAtual.Month;
int ano = dataAtual.Year;

Console.WriteLine($"Dia: {dia}");
Console.WriteLine($"Mês: {mes}");
Console.WriteLine($"Ano: {ano}");
```

Nesse exemplo, `DateTime.Now` é usado para obter a data e a hora atuais. As propriedades `Day`, `Month` e `Year` da classe `DateTime` são usadas para obter o dia, mês e ano, respectivamente.

Em C#, a conversão de dados de string para outros tipos pode ser feita através de vários métodos. Aqui estão alguns exemplos:

1. **Método Parse()**: Este método tenta converter a string para o tipo especificado. Se a conversão não for possível, ele lançará uma exceção. Aqui está um exemplo de como usar o método Parse() para converter uma string em um inteiro e um double:

```
string str1 = "10";
int numero1 = int.Parse(str1);
Console.WriteLine(numero1);

string str2 = "3.14";
double numero2 = double.Parse(str2);
Console.WriteLine(numero2);
```

1. **Método TryParse()**: Este método tenta converter a string para o tipo especificado, e retorna um valor booleano indicando se a conversão foi bem-sucedida ou não. Ao contrário do Parse(), ele não lançará uma exceção se a conversão falhar. Aqui está um exemplo de como usar o método TryParse():

```
string str1 = "100";
int numero1;
bool sucesso1 = int.TryParse(str1, out numero1);
if (sucesso1)
{
    Console.WriteLine(numero1);
}
else
{
    Console.WriteLine("A conversão falhou.");
}

string str2 = "3,14";
double numero2;
bool sucesso2 = double.TryParse(str2, out numero2);
if (sucesso2)
{
    Console.WriteLine(numero2);
}
else
{
    Console.WriteLine("A conversão falhou.");
}
```

1. **Método Convert.ToType()**: A classe Convert fornece vários métodos estáticos para converter um tipo de valor base em outro tipo de valor base. A conversão falhará se o valor não puder ser convertido para o tipo desejado. Aqui está um exemplo de como usar o método Convert.ToInt32() e Convert.ToDouble():

```
string str1 = "200";
int numero1 = Convert.ToInt32(str1);
Console.WriteLine(numero1);

string str2 = "6.28";
double numero2 = Convert.ToDouble(str2);
Console.WriteLine(numero2);
```

Lembre-se de que é importante sempre verificar se a string pode ser convertida para o tipo desejado antes de tentar a conversão, para evitar exceções em tempo de execução no seu código.

Criando um Projeto Console Application no Visual Studio Community

1. Abra o Visual Studio Community.
2. Clique em "Arquivo" na barra de menus, em seguida, selecione "Novo" e depois "Projeto". Isso abrirá a janela "Criar um novo projeto".
3. Na janela "Criar um novo projeto", digite "Aplicativo de console" na caixa de pesquisa e pressione "Enter". Isso filtrará a lista de templates de projeto.
4. Selecione "Aplicativo de console (.NET Core)" na lista de templates.
5. Clique em "Próximo".
6. Na próxima página, digite o nome do seu projeto no campo "Nome do projeto". Você também pode escolher onde salvar seu projeto clicando no botão "Procurar" ao lado do campo "Localização".
7. Clique em "Criar".
8. O Visual Studio criará o projeto e abrirá o arquivo Program.cs em uma nova janela. Você agora está pronto para começar a codificar seu aplicativo de console.

Configurando o Visual Studio Code para criar e rodar um projeto Console Application

1. Baixe e instale o Visual Studio Code, se ainda não o fez. Você pode baixá-lo do site oficial: https://code.visualstudio.com/.
2. Abra o Visual Studio Code.
3. Instale a extensão C# para Visual Studio Code. Você pode fazer isso procurando por 'C#' na barra de pesquisa na seção de extensões (acessível no menu à esquerda) e clicando em 'Instalar' na extensão C# fornecida pela Microsoft.
4. Instale o .NET Core SDK em seu computador. Isso permitirá que você crie e execute projetos .NET Core, incluindo aplicativos de console.

Você pode baixá-lo do site oficial .NET: https://dotnet.microsoft.com/download.

5. Após instalar o .NET Core SDK, você pode criar um novo projeto Console Application. Para isso, abra o terminal no Visual Studio Code (View > Terminal) e navegue até a pasta onde você deseja criar o projeto. Em seguida, digite o seguinte comando:

   ```
   dotnet new console -n NomeDoSeuProjeto
   ```

   Este comando criará um novo projeto Console Application com o nome que você especificar.

6. Navegue até a pasta do projeto recém-criado no terminal usando o comando 'cd':

   ```
   cd NomeDoSeuProjeto
   ```

7. Agora você pode executar o projeto digitando o seguinte comando no terminal:

   ```
   dotnet run
   ```

   Este comando compila e executa o projeto. Você deve ver a saída 'Hello World!' no terminal, que é a saída padrão para um novo projeto Console Application.

8. Você está pronto para começar a codificar! Você pode abrir o arquivo 'Program.cs' no explorador de arquivos do Visual Studio Code (acessível no menu à esquerda) e começar a editar seu código.

Lembre-se de salvar seu arquivo (File > Save) antes de executar o comando 'dotnet run' para ver as alterações no seu código.

Exercício 1:
Crie uma classe chamada `Carro` com as seguintes propriedades e métodos:

- Propriedades:
  - `Marca`

    - `Modelo`
    - `Ano`
    - `Cor`
- Métodos:
    - `Ligar()`
    - `Desligar()`
    - `Acelerar()`
    - `Frear()`

Depois, instancie dois objetos dessa classe e chame seus métodos para testá-los.

```
class Carro
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
    public int Ano { get; set; }
    public string Cor { get; set; }

    public void Ligar()
    {
        Console.WriteLine($"{Marca} {Modelo} está ligad
o.");
    }

    public void Desligar()
    {
        Console.WriteLine($"{Marca} {Modelo} está deslig
ado.");
    }

    public void Acelerar()
    {
        Console.WriteLine($"{Marca} {Modelo} está aceler
ando.");
    }
```

```
    public void Frear()
    {
        Console.WriteLine($"{Marca} {Modelo} está freian
do.");
    }
}

class Program
{
    static void Main()
    {
        Carro carro1 = new Carro { Marca = "Ford", Model
o = "Mustang", Ano = 2020, Cor = "Vermelho" };
        carro1.Ligar();
        carro1.Acelerar();
        carro1.Frear();
        carro1.Desligar();

        Carro carro2 = new Carro { Marca = "Chevrolet",
Modelo = "Camaro", Ano = 2021, Cor = "Amarelo" };
        carro2.Ligar();
        carro2.Acelerar();
        carro2.Frear();
        carro2.Desligar();
    }
}
```

Exercício 2:
Crie uma classe
`ContaBancaria` com as seguintes propriedades e métodos:

- Propriedades:
  - `NumeroDaConta`
  - `NomeDoTitular`
  - `Saldo`
- Métodos:

- `Depositar(valor)`
- `Sacar(valor)`

Depois, instancie um objeto dessa classe, faça depósitos e saques e imprima o saldo atual.

```
class ContaBancaria
{
    public string NumeroDaConta { get; set; }
    public string NomeDoTitular { get; set; }
    public double Saldo { get; private set; }

    public void Depositar(double valor)
    {
        Saldo += valor;
        Console.WriteLine($"Depósito de {valor:C} realiz
ado. Saldo atual: {Saldo:C}.");
    }

    public void Sacar(double valor)
    {
        if (valor <= Saldo)
        {
            Saldo -= valor;
            Console.WriteLine($"Saque de {valor:C} reali
zado. Saldo atual: {Saldo:C}.");
        }
        else
        {
            Console.WriteLine("Saldo insuficiente para r
ealizar o saque.");
        }
    }
}

class Program
{
    static void Main()
    {
```

```
        ContaBancaria conta = new ContaBancaria { Numero
DaConta = "123456", NomeDoTitular = "João Silva" };
        conta.Depositar(1000.0);
        conta.Sacar(200.0);
        conta.Sacar(1000.0);
    }
}
```

Exercício 3:
Crie um sistema agrícola utilizando os conceitos de Programação Orientada a Objetos em C#. Neste sistema, você deve incluir uma classe base `Plantação` e duas classes filhas `Milho` e `Soja`.

- Classe `Plantação`:
  - Propriedades: `Nome`, `Quantidade`
  - Métodos: `Plantar()`, `Colher()`
- Classe `Milho` (herda de `Plantação`):
  - Método: `Fertilizar()`
- Classe `Soja` (herda de `Plantação`):
  - Método: `Irrigar()`

Depois, instancie objetos dessas classes e chame seus métodos para testá-los.

```
class Plantacao
{
    public string Nome { get; set; }
    public int Quantidade { get; set; }

    public void Plantar()
    {
        Console.WriteLine($"{Nome} está sendo plantad
o.");
    }

    public void Colher()
    {
```

```
        Console.WriteLine($"{Nome} está sendo colhid
o.");
    }
}

class Milho : Plantacao
{
    public void Fertilizar()
    {
        Console.WriteLine("O milho está sendo fertilizad
o.");
    }
}

class Soja : Plantacao
{
    public void Irrigar()
    {
        Console.WriteLine("A soja está sendo irrigad
a.");
    }
}

class Program
{
    static void Main()
    {
        Milho milho = new Milho { Nome = "Milho", Quanti
dade = 200 };
        milho.Plantar();
        milho.Fertilizar();
        milho.Colher();

        Soja soja = new Soja { Nome = "Soja", Quantidade
= 300 };
        soja.Plantar();
        soja.Irrigar();
        soja.Colher();
```

```
    }
}
```