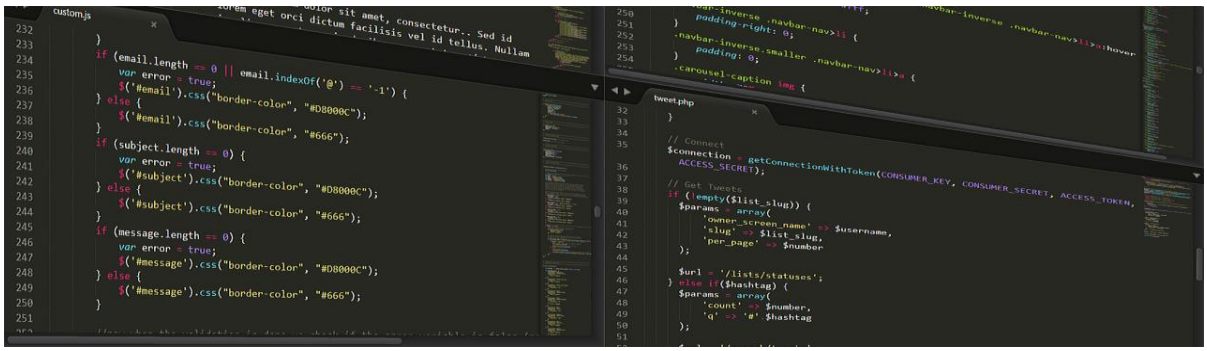
FUNDAÇÃO FAT

Programação Orientada a Objetos

Desenvolvedor de *Salesforce*



Sumário

Apresentação

Sejam bem-vindos a aula de nivelamento de conceitos de Programação Orientada a Objetos, preparatório para o *Salesforce Platform Developer Credential.* Neste curso, iremos explorar os principais conceitos e técnicas da Programação Orientada a Objetos (POO) utilizando a linguagem C#. Os conceitos a serem abordados são:

Classes e Objetos: A base da POO, onde classes são modelos para criar objetos que possuem atributos e comportamentos.

Encapsulamento: O princípio de esconder detalhes internos de uma classe e permitir acesso controlado aos seus membros.

Herança: A capacidade de uma classe de herdar atributos e métodos de outra classe, facilitando a reutilização de código.

Polimorfismo: A capacidade de um objeto ser tratado de múltiplas formas, permitindo o uso de métodos com o mesmo nome em classes diferentes.

Interfaces: Contratos que definem um conjunto de métodos e propriedades que as classes devem implementar.

Abstração: O uso de classes abstratas e interfaces para definir comportamentos sem especificar a implementação.

Sobrecarga de Métodos e Operadores: A capacidade de ter métodos com o mesmo nome, mas com diferentes parâmetros, e sobrecarregar operadores para classes personalizadas.

1. Introdução à Programação Orientada a Objetos (POO)

A **Programação Orientada a Objetos** (**POO**) é um paradigma de programação que utiliza "objetos" para representar dados e métodos. A **POO** é fundamental para o desenvolvimento de software moderno, pois facilita a organização, a reutilização e a manutenção do código.

1.1. O Que São Objetos e Classes?

Um **objeto** é uma instância de uma classe. Pense em um objeto como uma entidade que possui **características** (**atributos**) e **comportamentos** (**métodos**). Por exemplo, um carro pode ser um objeto que tem atributos como cor, marca e modelo, e comportamentos como acelerar, frear e buzinar.

Uma **classe** é um modelo ou uma planta que define as características e comportamentos que os objetos criados a partir dela terão. Em termos de programação, uma classe define um tipo de dado abstrato que agrupa **atributos** (**variáveis**) e **métodos** (**funções**).

Exemplo em C#:

```
public class Carro
{
    // Atributos
    public string Cor;
    public string Marca;
    public string Modelo;

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

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

Neste exemplo, **Carro** é uma **classe** que possui **atributos** (**Cor**, **Marca**, **Modelo**) e **métodos** (**Acelerar**, **Frear**).

1.2. Métodos e Propriedades

Os **métodos** são funções definidas dentro de uma classe que descrevem os comportamentos dos objetos dessa classe. Eles podem realizar ações ou operações nos atributos do objeto ou interagir com outros objetos.

Exemplo em C#:

```
public void Buzinar()
{
    Console.WriteLine("O carro está buzinando.");
}
```

As **propriedades** são uma forma de acessar os atributos de uma classe. Elas podem ser usadas para ler e escrever valores dos atributos, proporcionando um controle maior sobre como esses atributos são acessados e modificados.

Exemplo em C#:

```
private string _cor;

public string Cor
{
    get { return _cor; }
    set { _cor = value; }
}
```

1.3. Diferenciação da POO de Outros Paradigmas de Programação

Na **Programação Procedural**, o foco está na execução de sequências de comandos ou procedimentos. O código é organizado em funções ou sub-rotinas que realizam tarefas específicas. Este paradigma é eficiente para programas pequenos e simples, mas pode se tornar difícil de gerenciar e manter à medida que o programa cresce.

Exemplo de Código Procedural:

```
public void ProcessarPagamento()
{
    // Código para processar o pagamento
}
```

A **Programação Orientada a Objetos (POO)**, por outro lado, organiza o código em torno de objetos e classes. Isso permite que o código seja modular, reutilizável e mais fácil de manter. A **POO** também facilita a modelagem de problemas do mundo real de maneira mais intuitiva.

1.4. Benefícios da Abordagem Orientada a Objetos

- **Organização:** A **POO** organiza o código em classes e objetos, tornando-o mais modular e estruturado.
- **Reutilização**: A herança permite que classes compartilhem código e funcionalidades, promovendo a reutilização.
- **Manutenção**: O encapsulamento oculta os detalhes internos das classes, facilitando a manutenção e a modificação do código sem afetar outras partes do programa.
- **Flexibilidade**: O polimorfismo permite que o mesmo código funcione com diferentes tipos de objetos, aumentando a flexibilidade e a capacidade de extensão do software.

Conclusão

Compreender os conceitos básicos da POO é crucial para qualquer desenvolvedor moderno. A **POO** não só melhora a organização e a legibilidade do código, mas também promove práticas de programação que facilitam a reutilização e a manutenção do software. Nos próximos capítulos, exploraremos esses conceitos detalhadamente e veremos como implementá-los em **C#**.

2. Classes e Objetos em C#

No capítulo anterior, introduzimos os conceitos básicos da **Programação Orientada a Objetos** (**POO**). Neste capítulo, vamos aprofundar nosso entendimento sobre como definir classes e criar objetos em **C#**. Vamos explorar a sintaxe específica de **C#** para declarar atributos, métodos construtores, métodos de instância, propriedades e campos. Também veremos técnicas de inicialização de objetos.

2.1. Definindo Classes em C#

Uma **classe** em **C#** é definida usando a palavra-chave class seguida pelo nome da classe. Dentro da classe, podemos declarar atributos, métodos e propriedades.

Exemplo de uma classe simples:

```
public class Carro
{
    // Atributos
    public string Cor;
    public string Marca;
    public string Modelo;

    // Método de instância
    public void Acelerar()
    {
        Console.WriteLine("O carro está acelerando.");
    }

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

Neste exemplo, **Carro** é uma classe que define **três atributos** (Cor, Marca, Modelo) e **dois métodos** (Acelerar, Frear).

2.2. Criando Objetos

Para criar um objeto de uma classe, usamos a palavra-chave new, que invoca o construtor da classe.

Exemplo de criação de objeto:

```
Carro meuCarro = new Carro();
meuCarro.Cor = "Vermelho";
meuCarro.Marca = "Toyota";
meuCarro.Modelo = "Corolla";

meuCarro.Acelerar();
```

Neste exemplo, criamos um objeto **meuCarro** da classe **Carro** e definimos seus atributos. Em seguida, chamamos o método **Acelerar**.

2.3. Métodos Construtores

Os métodos construtores são usados para inicializar objetos. Eles têm o mesmo nome da classe e não possuem tipo de retorno.

Exemplo de construtor:

```
public class Carro
{
    public string Cor;
    public string Marca;
    public string Modelo;

    // Construtor
    public Carro(string cor, string marca, string modelo)
    {
        Cor = cor;
        Marca = marca;
        Modelo = modelo;
    }

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

Uso do construtor:

```
Carro meuCarro = new Carro("Vermelho", "Toyota", "Corolla");
meuCarro.Acelerar();
```

2.4. Propriedades

As propriedades em C# permitem um controle mais refinado sobre o acesso aos atributos de uma classe. Elas podem incluir lógica de validação e outras operações.

Exemplo de propriedades:

```
public class Carro
{
    private string cor;

    public string Cor
    {
        get { return cor; }
        set { cor = value; }
    }

    public string Marca { get; set; }
    public string Modelo { get; set; }

    public Carro(string cor, string marca, string modelo)
    {
        Cor = cor;
        Marca = marca;
        Modelo = modelo;
    }

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

2.5. Campos

Os campos são variáveis definidas dentro de uma classe. Eles são geralmente privados e acessados através de propriedades.

Exemplo de campos:

```
public class Carro
{
    private string cor;
    private string marca;
    private string modelo;

    public string Cor
    {
        get { return cor; }
        set { cor = value; }
    }

    public string Marca
    {
        get { return marca; }
        set { marca = value; }
    }

    public string Modelo
    {
        get { return modelo; }
        set { modelo = value; }
    }

    public Carro(string cor, string marca, string modelo)
    {
        this.cor = cor;
        this.marca = marca;
        this.modelo = modelo;
    }

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

2.6. Inicialização de Objetos

Em **C#**, podemos inicializar objetos de várias maneiras, incluindo inicializadores de objeto e usando construtores.

Inicializadores de Objeto:

```
Carro meuCarro = new Carro
{
    Cor = "Vermelho",
    Marca = "Toyota",
    Modelo = "Corolla"
};

meuCarro.Acelerar();
```

Neste capítulo, vimos como definir classes e criar objetos em **C#**. Aprendemos sobre atributos, métodos construtores, métodos de instância, propriedades e campos, e exploramos técnicas de inicialização de objetos. Esses conceitos são fundamentais para a **POO** e formam a base para construir aplicações robustas e reutilizáveis em **C#.** Nos próximos capítulos, continuaremos a expandir nosso conhecimento sobre **POO** e como aplicá-lo em **C#**.

3. Encapsulamento e Modificadores de Acesso

O encapsulamento é um princípio-chave da **POO** que visa proteger os detalhes internos de uma classe. Neste capítulo, aprenderemos sobre modificadores de acesso em **C#** (**public**, **private**, **protected**, etc.), encapsulamento de dados e métodos, e como usar propriedades para controlar o acesso aos membros de uma classe.

3.1. Introdução ao Encapsulamento

O encapsulamento é um dos pilares da **Programação Orientada a Objetos** (**POO**). Ele se refere à técnica de esconder os detalhes internos de uma classe e restringir o acesso aos seus componentes. Isso garante que os dados de um objeto sejam manipulados apenas através de métodos definidos, promovendo a segurança e a integridade dos dados.

3.2. Modificadores de Acesso em C#

Em **C#**, os modificadores de acesso controlam a visibilidade dos membros de uma classe (atributos, métodos, propriedades, etc.). Os principais modificadores de acesso são:

- **public**: O membro é acessível de qualquer lugar do código.
- **private**: O membro é acessível apenas dentro da própria classe.
- **protected**: O membro é acessível dentro da própria classe e em classes derivadas.
- **internal**: O membro é acessível apenas dentro do mesmo assembly (projeto).
- **protected internal**: O membro é acessível dentro do mesmo assembly ou em classes derivadas.

3.3. Encapsulamento de Dados e Métodos

O encapsulamento é realizado utilizando modificadores de acesso para proteger os dados e métodos de uma classe. Vamos ver um exemplo prático.

Exemplo em C#:

```
public class ContaBancaria
{
    // Atributos privados
    private double saldo;

    // Propriedade pública para acessar o saldo
    public double Saldo
    {
        get { return saldo; }
        private set { saldo = value; }
    }

    // Construtor
    public ContaBancaria(double saldoInicial)
    {
        saldo = saldoInicial;
    }

    // Método público para depositar dinheiro
    public void Depositar(double quantia)
    {
        if (quantia > 0)
        {
            saldo += quantia;
        }
    }

    // Método público para sacar dinheiro
    public void Sacar(double quantia)
    {
        if (quantia > 0 && quantia <= saldo)
        {
            saldo -= quantia;
        }
    }
}
```

Neste exemplo, a classe **ContaBancaria** encapsula o atributo **saldo**, tornando-o privado. O **saldo** só pode ser acessado e modificado através dos métodos **Depositar** e **Sacar**, que contêm lógica para garantir que as operações sejam válidas.

3.4. Propriedades para Controle de Acesso

As propriedades em **C#** são usadas para controlar o acesso aos atributos privados. Elas podem incluir lógica para validar dados antes de permitir que sejam alterados ou retornados.

Exemplo de uso de propriedades:

```
public class Pessoa
{
    private int idade;

    public int Idade
    {
        get { return idade; }
        set
        {
            if (value >= 0)
            {
                idade = value;
            }
        }
    }

    public Pessoa(int idadeInicial)
    {
        Idade = idadeInicial;
    }
}
```

Neste exemplo, a propriedade **Idade** encapsula o atributo **idade**. A lógica no método **set** garante que a idade não possa ser definida para um valor negativo.

Exemplo Completo

Vamos juntar tudo o que aprendemos em um exemplo mais completo.

```
public class Produto
{
    // Atributos privados
    private string nome;
    private double preco;

    // Propriedades públicas
    public string Nome
    {
        get { return nome; }
        set
        {
            if (!string.IsNullOrWhiteSpace(value))
            {
                nome = value;
            }
        }
    }
    public double Preco
    {
        get { return preco; }
        set
        {
            if (value > 0)
            {
                preco = value;
            }
        }
    }

    // Construtor
    public Produto(string nome, double preco)
    {
        Nome = nome;
        Preco = preco;
    }

    // Método público
    public void MostrarDetalhes()
    {
        Console.WriteLine($"Produto: {Nome}, Preço: {Preco:C}");
    }
}
```

```
class Program
{
    static void Main()
    {
        Produto produto = new Produto("Laptop", 1500.00);
        produto.MostrarDetalhes();

        // Tentativa de definir um preço negativo - será ignorada
        produto.Preco = -500.00;
        produto.MostrarDetalhes();
    }
}
```

Neste exemplo, a classe Produto encapsula os atributos nome e preco, permitindo acesso e modificação apenas através das propriedades Nome e Preco, que contêm lógica para validação.

Conclusão

O encapsulamento é um princípio fundamental da **POO** que promove a segurança e a integridade dos dados em um programa. Usando modificadores de acesso e propriedades, podemos controlar como os dados são acessados e modificados, garantindo que as operações sejam realizadas de maneira segura e correta. No próximo capítulo, exploraremos a herança, que permite criar novas classes baseadas em classes existentes.

4. Encapsulamento e Modificadores de Acesso

A **herança** e o **polimorfismo** são conceitos fundamentais da **Programação Orientada a Objetos** (**POO**) que permitem a criação de hierarquias de classes e o tratamento uniforme de objetos de diferentes classes. Neste capítulo, aprenderemos como implementar herança em **C#**, utilizar classes base e derivadas, e explorar o conceito de polimorfismo.

4.1. Herança

A **herança** é um mecanismo que permite que uma classe (chamada de classe derivada ou subclasse) herde membros (atributos e métodos) de outra classe (chamada de classe base ou superclasse). Isso promove a reutilização de código e estabelece uma relação "é-um" entre as classes.

Para implementar a herança em **C#**, usamos o símbolo **:** após o nome da classe derivada, seguido pelo nome da classe base.

Exemplo em C#:

```
public class Animal
{
    public string Nome { get; set; }

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

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

Neste exemplo, a classe **Cachorro** herda da classe **Animal**, o que significa que **Cachorro** possui todos os membros de **Animal** além dos seus próprios.

4.2. Usando Classes Base e Derivadas

Ao criar uma instância de uma classe derivada, ela terá acesso a todos os membros da classe base.

Exemplo de uso:

```
Cachorro meuCachorro = new Cachorro();
meuCachorro.Nome = "Rex";
meuCachorro.Comer();  // Saída: Rex está comendo.
meuCachorro.Latir();  // Saída: Rex está latindo.
```

4.3. Polimorfismo

O **polimorfismo** permite que objetos de diferentes classes sejam tratados de forma uniforme. Em **C#**, isso é geralmente realizado através da herança e do uso de métodos virtuais e sobrecarregados. Para permitir que um método em uma classe base seja substituído por um método em uma classe derivada, usamos a palavra-chave **virtual** na declaração do método na classe base e **override** na classe derivada.

Exemplo em C#:

```
public class Animal
{
    public string Nome { get; set; }

    public virtual void EmitirSom()
    {
        Console.WriteLine($"{Nome} está emitindo um som.");
    }
}

public class Cachorro : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine($"{Nome} está latindo.");
    }
}
```

Neste exemplo, **EmitirSom** é um **método virtual** na classe **Animal** e é **substituído** (**override**) na classe **Cachorro**.

Tratando Objetos de Forma Uniforme

Graças ao polimorfismo, podemos tratar objetos de diferentes classes derivadas como se fossem da classe base.

Exemplo de polimorfismo:

```
Animal meuAnimal = new Cachorro();
meuAnimal.Nome = "Rex";
meuAnimal.EmitirSom();  // Saída: Rex está latindo.
```

Neste exemplo, embora **meuAnimal** seja declarado como do tipo **Animal**, ele referencia um objeto **Cachorro**, e o método sobrescrito **EmitirSom** da classe **Cachorro** é chamado.

4.4. Classes Abstratas

Uma **classe abstrata** é uma classe que não pode ser instanciada diretamente. Ela serve como uma base para outras classes. **Métodos** em uma classe abstrata podem ser definidos como abstratos, indicando que devem ser implementados nas classes derivadas.

Exemplo de classe abstrata:

```
public abstract class Animal
{
    public string Nome { get; set; }
    public abstract void EmitirSom();
}

public class Cachorro : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine($"{Nome} está latindo.");
    }
}
```

Neste exemplo, **Animal** é uma **classe abstrata** com um **método abstrato EmitirSom** que deve ser implementado pela classe derivada **Cachorro**.

4.5. Interfaces

Interfaces definem um contrato que uma classe deve cumprir. Elas especificam métodos e propriedades que devem ser implementados, mas não fornecem a implementação.

Exemplo de interface:

```
public interface IAnimal
{
    string Nome { get; set; }
    void EmitirSom();
}

public class Gato : IAnimal
{
    public string Nome { get; set; }

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

Neste exemplo, a interface **IAnimal** define o contrato que a classe **Gato** deve implementar.

Conclusão

A **herança** e o **polimorfismo** são ferramentas poderosas na **POO** que permitem a reutilização de código e o tratamento uniforme de objetos de diferentes classes. A herança permite criar hierarquias de classes, enquanto o polimorfismo permite que métodos definidos na classe base sejam substituídos nas classes derivadas. Entender esses conceitos é crucial para construir aplicações robustas e flexíveis em **C#**. No próximo capítulo, exploraremos interfaces e classes abstratas em mais detalhes.

5. Revisão

Neste capítulo, faremos uma revisão dos principais conceitos da **Programação Orientada a Objetos** (**POO**) que abordamos nos capítulos anteriores. Vamos recapitular os fundamentos da **POO**, como classes e objetos, encapsulamento, herança, polimorfismo, e mais. Este capítulo servirá como um guia de referência rápida para consolidar o aprendizado.

Conceitos Básicos da POO:

- **Classe**: Uma classe é uma planta ou modelo que define as características e comportamentos de um tipo de objeto.
- **Objeto**: Um objeto é uma instância de uma classe. Ele possui atributos e métodos definidos pela classe.

Exemplo:

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

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

Carro meuCarro = new Carro();
meuCarro.Cor = "Vermelho";
meuCarro.Marca = "Toyota";
meuCarro.Acelerar();
```

Encapsulamento:

- **Encapsulamento**: Protege os detalhes internos de uma classe e controla o acesso aos seus membros através de modificadores de acesso.
- **Modificadores de Acesso**: *public, private, protected, internal, protected internal.*

Exemplo:

```
public class ContaBancaria
{
    private double saldo;

    public double Saldo
    {
        get { return saldo; }
        private set { saldo = value; }
    }

    public void Depositar(double quantia)
    {
        if (quantia > 0)
        {
            saldo += quantia;
        }
    }

    public void Sacar(double quantia)
    {
        if (quantia > 0 && quantia <= saldo)
        {
            saldo -= quantia;
        }
    }
}
```

Herança:

- **Herança**: Permite que uma classe (derivada) herde atributos e métodos de outra classe (base).
- **Classe Base**: A classe da qual outras classes derivam.
- **Classe Derivada:** A classe que herda de outra classe.

Exemplo:

```
public class Animal
{
    public string Nome { get; set; }

    public virtual void EmitirSom()
    {
        Console.WriteLine($"{Nome} está emitindo um som.");
    }
}

public class Cachorro : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine($"{Nome} está latindo.");
    }
}

Animal meuAnimal = new Cachorro();
meuAnimal.Nome = "Rex";
meuAnimal.EmitirSom();
```

Polimorfismo:

- **Polimorfismo**: Permite que objetos de diferentes classes derivadas sejam tratados de forma uniforme através de uma interface comum.
- **Métodos Virtuais e Override**: Métodos na classe base podem ser substituídos em classes derivadas para fornecer comportamento específico.

Exemplo:

```
public class Gato : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine($"{Nome} está miando.");
    }
}

Animal meuCachorro = new Cachorro();
Animal meuGato = new Gato();

meuCachorro.Nome = "Rex";
meuGato.Nome = "Mimi";

meuCachorro.EmitirSom();  // Saída: Rex está latindo.
meuGato.EmitirSom();  // Saída: Mimi está miando.
```

Classes Abstratas e Interfaces:

- **Classe Abstrata**: Uma classe que não pode ser instanciada diretamente e serve como base para outras classes. Pode conter métodos abstratos que devem ser implementados nas classes derivadas.
- **Interface**: Define um contrato que uma classe deve cumprir, especificando métodos e propriedades, mas sem fornecer a implementação.

Exemplo de Classe Abstrata:

```
public abstract class Animal
{
    public string Nome { get; set; }
    public abstract void EmitirSom();
}

public class Cachorro : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine($"{Nome} está latindo.");
    }
}
```

Exemplo de Interface:

```
public interface IAnimal
{
    string Nome { get; set; }
    void EmitirSom();
}

public class Gato : IAnimal
{
    public string Nome { get; set; }

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

6. Conclusão

Neste material, você teve uma introdução sólida aos fundamentos da Programação Orientada a Objetos usando C#. Agora, você deve ter um entendimento claro de como criar classes, usar herança, interfaces, abstração e trabalhar com conceitos avançados como sobrecarga de métodos e operadores.

É importante praticar esses conceitos através de projetos práticos e desafios para fortalecer seu conhecimento e habilidades. Continue explorando recursos adicionais, como padrões de projeto e boas práticas de desenvolvimento, para se tornar um programador mais completo e eficiente.

Lembre-se sempre de que a prática constante e a busca por aprendizado são fundamentais para o sucesso na programação. Estamos ansiosos para ver suas conquistas e projetos futuros!

Exercícios Práticos

Com base nos conceitos estudados, desenvolva os códigos necessários a resolução das solicitações abaixo, de acordo com cada capítulo estudado. A resolução de cada parte encontra-se ao final do material.

Capítulo 1 – Introdução

1. Crie uma classe **Pessoa** com os atributos **Nome** e **Idade**, e um método **MostrarInformacoes** que exiba esses atributos.
2. Modifique a classe **Pessoa** para incluir um **construtor** que inicialize os atributos **Nome** e **Idade** na criação do objeto.
3. Crie um objeto **Pessoa** e chame o método **MostrarInformacoes**.

Capítulo 2 – Classes e Objetos em C#

1. Modifique a classe **Pessoa** do exercício anterior para usar **propriedades** em vez de **atributos públicos**.
2. Crie um método na classe **Pessoa** chamado **FazerAniversario** que incrementa a idade em 1 unidade.
3. Crie um objeto **Pessoa**, altere sua **idade** usando o método **FazerAniversario** e exiba as informações da pessoa.

Capítulo 3 – Encapsulamento e Modificadores de Acesso

1. Crie uma classe **ContaBancaria** com um atributo privado **saldo** e métodos **Depositar** e **Sacar** que alteram o saldo.
2. Modifique a classe **ContaBancaria** para usar encapsulamento, permitindo acesso controlado ao saldo.
3. Crie um método **MostrarSaldo** na classe **ContaBancaria** que exibe o saldo da conta.

Capítulo 4 – Herança e Polimorfismo

1. Crie uma classe **FormaGeometrica** com um método **CalcularArea** e um método **ExibirTipo** que exibe o tipo da forma.
2. Crie classes derivadas **Quadrado** e **Circulo** que herdam de **FormaGeometrica** e implementam seus próprios métodos **CalcularArea**.

3. Crie um método **ExibirDetalhes** na classe **FormaGeometrica** que chama os métodos **ExibirTipo** e **CalcularArea**.

Capítulo 4 – Interfaces e Abstração

1. Crie uma interface **IAutenticavel** com um método **Autenticar** que retorna verdadeiro se a autenticação for bem-sucedida.
2. Crie classes **Usuario** e **Administrador** que implementam **IAutenticavel** e o método **Autenticar**.
3. Crie um método **Login** na classe **Program** que recebe um objeto **IAutenticavel** e realiza a autenticação.

Resolução Capítulo 1 – Introdução

```
using System;

// Definindo uma classe Pessoa
public class Pessoa
{
    // Atributos da classe
    public string Nome;
    public int Idade;

    // Construtor da classe
    public Pessoa(string nome, int idade)
    {
        Nome = nome;
        Idade = idade;
    }

    // Método da classe
    public void MostrarInformacoes()
    {
        Console.WriteLine("Nome: " + Nome);
        Console.WriteLine("Idade: " + Idade);
    }
}

class Program
{
    static void Main()
    {
        // Criando um objeto Pessoa com construtor
        Pessoa pessoa1 = new Pessoa("Maria", 30);

        // Chamando o método MostrarInformacoes
        pessoa1.MostrarInformacoes();
    }
}
```

Resolução Capítulo 2 – Classes e Objetos em C#

```
using System;

// Definindo uma classe Pessoa com propriedades
public class Pessoa
{
    // Propriedades da classe
    public string Nome { get; set; }
    public int Idade { get; private set; }

    // Construtor da classe
    public Pessoa(string nome, int idade)
    {
        Nome = nome;
        Idade = idade;
    }

    // Método para fazer aniversário
    public void FazerAniversario()
    {
        Idade++;
    }

    // Método da classe
    public void MostrarInformacoes()
    {
        Console.WriteLine("Nome: " + Nome);
        Console.WriteLine("Idade: " + Idade);
    }
}

class Program
{
    static void Main()
    {
        // Criando um objeto Pessoa com construtor
        Pessoa pessoa1 = new Pessoa("João", 25);

        // Chamando o método FazerAniversario
        pessoa1.FazerAniversario();

        // Exibindo as informações da pessoa
        pessoa1.MostrarInformacoes();
    }
}
```

Resolução Capítulo 3 – Encapsulamento e Modificadores de Acesso

```
using System;

// Definindo uma classe ContaBancaria com encapsulamento
public class ContaBancaria
{
    // Atributo privado
    private double saldo;

    // Propriedade pública para acesso controlado ao saldo
    public double Saldo
    {
        get { return saldo; }
        private set { saldo = value; }
    }

    // Construtor da classe
    public ContaBancaria(double saldoInicial)
    {
        Saldo = saldoInicial;
    }

    // Método para depositar dinheiro na conta
    public void Depositar(double valor)
    {
        Saldo += valor;
        Console.WriteLine("Depósito de R$" + valor + " realizado. Novo saldo: R$" +
Saldo);
    }

    // Método para sacar dinheiro da conta
    public void Sacar(double valor)
    {
        if (Saldo >= valor)
        {
            Saldo -= valor;
            Console.WriteLine("Saque de R$" + valor + " realizado. Novo saldo: R$"
+ Saldo);
        }
        else
        {
            Console.WriteLine("Saldo insuficiente para saque.");
        }
    }

    // Método para mostrar o saldo da conta
    public void MostrarSaldo()
    {
        Console.WriteLine("Saldo atual: R$" + Saldo);
    }
}
```

```
class Program
{
    static void Main()
    {
        // Criando uma conta bancária
        ContaBancaria minhaConta = new ContaBancaria(1000);

        // Fazendo operações na conta
        minhaConta.Depositar(500);
        minhaConta.Sacar(200);
        minhaConta.MostrarSaldo();
    }
}
```

Resolução Capítulo 4 – Herança e Polimorfismo

```
using System;

// Definindo uma classe FormaGeometrica (classe base)
public class FormaGeometrica
{
    public virtual void CalcularArea()
    {
        Console.WriteLine("Área calculada.");
    }

    public void ExibirTipo()
    {
        Console.WriteLine("Tipo da forma geométrica.");
    }

    public void ExibirDetalhes()
    {
        ExibirTipo();
        CalcularArea();
    }
}

// Definindo uma classe Quadrado (classe derivada de FormaGeometrica)
public class Quadrado : FormaGeometrica
{
    public override void CalcularArea()
    {
        Console.WriteLine("Área do quadrado calculada.");
    }
}

// Definindo uma classe Circulo (classe derivada de FormaGeometrica)
public class Circulo : FormaGeometrica
{
    public override void CalcularArea()
    {
        Console.WriteLine("Área do círculo calculada.");
    }
}

class Program
{
    static void Main()
    {
        // Criando objetos Quadrado e Circulo
        Quadrado meuQuadrado = new Quadrado();
        Circulo meuCirculo = new Circulo();

        // Chamando o método ExibirDetalhes para cada objeto
        meuQuadrado.ExibirDetalhes();
        meuCirculo.ExibirDetalhes();
    }
}
```

Resolução Capítulo 4 – Interfaces e Abstração

```
using System;

// Definindo uma interface IAutenticavel
public interface IAutenticavel
{
    bool Autenticar(string usuario, string senha);
}

// Definindo classes Usuario e Administrador que implementam IAutenticavel
public class Usuario : IAutenticavel
{
    public bool Autenticar(string usuario, string senha)
    {
        // Lógica de autenticação para usuário
        return true;
    }
}

public class Administrador : IAutenticavel
{
    public bool Autenticar(string usuario, string senha)
    {
        // Lógica de autenticação para administrador
        return true;
    }
}

class Program
{
    static void Login(IAutenticavel usuario)
    {
        bool autenticado = usuario.Autenticar("meuUsuario", "minhaSenha");

        if (autenticado)
        {
            Console.WriteLine("Login bem-sucedido.");
        }
        else
        {
            Console.WriteLine("Falha na autenticação.");
        }
    }

    static void Main()
    {
        // Criando objetos Usuario e Administrador
        Usuario meuUsuario = new Usuario();
        Administrador meuAdmin = new Administrador();

        // Chamando o método Login com diferentes objetos
        Login(meuUsuario);
        Login(meuAdmin);
    }
}
```