Programação Orientada a Objetos

Módulo 3

Herança, Classes Abstratas, Polimorfismo, Interfaces

Curso de Gestão da Informação

Prof. Dr. Daniel A. Furtado - FACOM/UFU

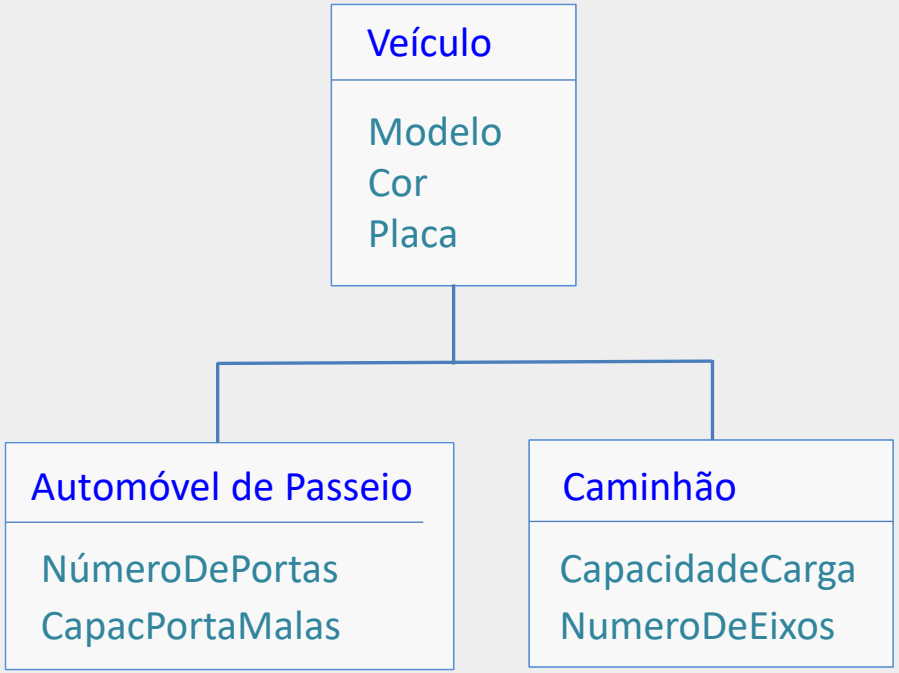
Objetivos do Módulo

- Apresentar o conceito de Generalização/Especialização
- Apresentar o conceito de Herança em POO
- Apresentar o conceito de Interfaces
- Introduzir o conceito de Polimorfismo

Conceito de Generalização / Especialização

- Generalização é o processo de identificar características e comportamentos comuns em várias entidades e abstraí-los para uma entidade genérica.
  - Automóveis, caminhões e motos possuem características e comportamentos comuns (cor, placa, ano, acelerar) e podem ser generalizados em veículos.
  - Lógica de pensamento "de baixo para cima".
- Especialização é o processo contrário, que parte de uma entidade mais ampla (como veiculo) e busca identificar características e comportamentos específicos de subgrupos.
  - Automóveis de passeio possuem um número de portas, uma capacidade do porta-malas.
  - Caminhões possuem uma capacidade de carga, um número de eixos.
  - Lógica de pensamento "de cima para baixo".

Conceito de Generalização / Especialização



- Neste exemplo, Automóvel de Passeio e Caminhão são especializações de Veículo, pois são tipos específicos de veículos, com características adicionais.
- Por outro lado, Veículo é uma generalização de Automóvel e Caminhão.

Herança

- Herança é um mecanismo que permite que uma classe (a classe derivada ou subclasse) adquira os atributos, propriedades e métodos de outra classe (a classe base).
- A herança modela uma relação do tipo "é um", permitindo implementar o conceito de generalização / especialização:
  - Um Automóvel é um Veiculo.
  - Um Cachorro é um Animal.
  - Um Funcionario é uma Pessoa.
- Por exemplo, é possível criar uma subclasse Caminhao, com características adicionais, a partir de uma classe base Veiculo; ou uma subclasse Funcionario a partir de uma classe base Pessoa.
- Denominações para a classe mais geral, sendo herdada:
  - Classe base, superclasse, classe pai
- Denominações para a classe mais específica, que herda
  - Classe derivada, subclasse, classe filha

Quais os Benefícios da Herança?

Reutilização de Código

- Atributos e métodos comuns são definidos uma única vez na classe base e reutilizados nas classes derivadas.

Organização e Manutenção

- Em muitos casos, subclasses específicas podem ser desenvolvidas e mantidas de forma independente da classe base.

Polimorfismo

- A herança é o alicerce para o polimorfismo, permitindo que objetos de tipos diferentes (Automóvel, Caminhão) sejam tratados de maneira uniforme (Veículo).

Herança - Sintaxe

- Para indicar a herança em C# utiliza-se o operador dois-pontos (:) na declaração da classe derivada:
  - class ClasseDerivada : ClasseBase
- Se a classe base tiver um construtor que exige parâmetros, a classe derivada precisa chamá-lo.
- A palavra-chave base é usada dentro de uma classe derivada para fazer referência a sua classe base. Seu uso mais comum é no construtor.

Herança - Exemplo

```
class Veiculo
{
    public string Modelo { get; }
    public Cor Cor { get; set; }
    public int Velocidade { get; set; }

    public Veiculo(string modelo) {
        Modelo = modelo;
        Velocidade = 0;
        Cor = CoresComuns.Azul;
    }
    public void Acelerar() ⇒ Velocidade++;
    public void Parar() ⇒ Velocidade = 0;
}
```

Definição da classe base Veiculo, que é mais geral e contém propriedades e métodos comuns a todo tipo de veículo. Repare que o construtor exige que o usuário informe apenas o modelo ao criar novos objetos do tipo Veiculo.

```
class Automovel : Veiculo
{
    public int NumeroDePortas { get; set; }
    public int CapacPortaMalas { get; set; }

    public Automovel(string modelo, int portas, int capMalas)
        : base(modelo)
    {
        NumeroDePortas = portas;
        CapacPortaMalas = capMalas;
    }
}
```

Definição da classe derivada Automovel, que é mais específica, herda as propriedades e métodos de Veiculo, e acrescenta as propriedades específicas de automóveis.

Repare que o construtor exige o fornecimento do modelo e dos dados específicos de automóvel. O código " : base(modelo)" faz a chamada do construtor da classe base Veiculo, repassando o modelo para o construtor da outra classe.

Herança - Exemplo

```
class Caminhao : Veiculo
{
    public int CapacCarga { get; set; }
    public int NumeroEixos { get; set; }

    public Caminhao(string modelo, int capacCarga, int eixos)
        : base(modelo)
    {
        CapacCarga = capacCarga;
        NumeroEixos = eixos;
    }
}
```

Definição da classe derivada Caminhao, que herda as propriedades e métodos de Veiculo, e acrescenta as propriedades específicas de caminhões. Repare que também foi inserido o código " : base(modelo)" para que o modelo seja repassado ao construtor da classe base Veiculo.

Herança - Exemplo

```
var moto1 = new Veiculo("Honda CG 160");
var carro1 = new Automovel("Volkswagen Polo", 4, 300);
var caminhao1 = new Caminhao("Volvo FH 450", 38000, 3);

carro1.Cor = CoresComuns.Preto; // Propriedade herdada de Veiculo
carro1.NumeroDePortas = 2;      // Propriedade própria de Automovel
carro1.Acelerar();              // Método herdado de Veiculo
Console.WriteLine(carro1.Velocidade); //Propriedade herdada de Veiculo

caminhao1.Cor = CoresComuns.Branco; // Propriedade herdada de Veiculo
caminhao1.CapacCarga = 40000;       // Propriedade própria de Caminhao
caminhao1.Acelerar();               // Método herdado de Veiculo
caminhao1.Parar();                  // Método herdado de Veiculo
Console.WriteLine(caminhao1.Velocidade); //Propriedade herdada de Veiculo
```

Herança - Observações

- Objetos da classe filha são sempre instanciados "de cima para baixo"
  - O construtor da classe base é sempre executado antes do construtor da classe filha.
- Portanto, se o construtor da classe base exige parâmetros, a classe filha é obrigada a chamá-lo utilizando a sintaxe : base(parametros).

O que Exatamente é Herdado?

- Membros da classe base definidos como private NÃO são herdados pela classe derivada.
  - O acesso continua restrito apenas ao código da própria classe.
- Membros da classe base definidos como public SÃO herdados.
- Com a herança surge um novo modificador de acesso: o protected.
  - Membros definidos como protected são similares aos private, porém com a liberação da herança, ou seja, PODEM ser acessados na classe derivada.
  - Porém, não podem ser acessados por código externo como no programa principal e outros objetos.

```
class Veiculo {
    private string Modelo { get; }
    public Cor Cor { get; set; }
    protected int Velocidade { get; set; }

    public Veiculo(string modelo) {
        Modelo = modelo;
        Velocidade = 0;
        Cor = CoresComuns.Azul;
    }
    public void Acelerar() ⇒ Velocidade++;
}
```

```
class Automovel : Veiculo {
    public int NumeroDePortas { get; set; }

    public Automovel(string modelo, int portas) : base(modelo) {
        NumeroDePortas = portas;
    }

    // A propriedade Velocidade definida em Veiculo
    // pode ser acessada aqui porque é protected
    public void AcelerarForte() ⇒ Velocidade += 10;
    public void Exibir() {
        // Console.Write(Modelo); Não permitido (Modelo é private)
        Cor.Descrever(); // Permitido (Cor é public)
        Console.Write(Velocidade); // Permitido (Vel. é protected)
    }
}
```

```
var carro1 = new Automovel("Volkswagen Polo", 4);

// Propriedade pública herdada de Veiculo
carro1.Cor = CoresComuns.Preto;

// Propriedade pública própria de Automovel
carro1.NumeroDePortas = 2;

// Método público herdado de Veiculo
carro1.Acelerar();

// Método público próprio de Automovel
carro1.AcelerarForte();

// Erro, pois Modelo é privado em Veiculo
carro1.Modelo = "Outro modelo";

// Erro, pois Velocidade é protected em Veiculo
carro1.Velocidade = 0;
```

Programa Principal

Exercício

- Defina uma classe para representar pessoas de uma forma geral, seja pessoa física ou pessoa jurídica. A classe deve ter uma propriedade pública Nome e um construtor que permita a criação de novos objetos da seguinte forma: `var pessoa1 = new Pessoa("Fulano de Tal");`
- Defina classes derivadas para representar pessoas físicas e jurídicas. Devem conter, respectivamente, um CPF e um CNPj. As classes devem ter construtores que permitam a criação de objetos da seguinte forma:
  - `var pessoaFisica1 = new PessoaFisica("Fulano de Tal", "cpf123");`
  - `var pessoaJuridica1 = new PessoaJuridica("Empresa X", "cnpj123");`
- Crie um "programa principal" exemplificando o uso de todas as classes e de suas propriedades.

Sobrescrita de Métodos

- Se for necessário, a classe filha pode fornecer a sua própria implementação para um método já definido na classe pai.
- Por exemplo, podemos criar um método Acelerar genérico na classe Veiculo que incrementa a velocidade em 10. Na classe Automovel poderíamos redefini-lo (sobrescreve-lo) para que a aceleração seja mais rápida (+20, por ex.) e na classe Caminhao, mais lenta (+5, por ex.).
- Para isso, o método Acelerar na classe pai precisa permitir a sobrescrita nas classes filhas. Isso é feito adicionando na definição a palavra virtual.
- Nas classes filhas, a nova versão do método precisa ser definida com a palavra override (sobrescrever) e não poderá alterar o tipo de retorno nem os tipos dos parâmetros.
- Este mecanismo é um exemplo de polimorfismo.
- Não confundir com sobrecarga de métodos.

Sobrescrita de Métodos

```
class Veiculo
{
    public string Modelo { get; }
    public int Veloc { get; set; } = 0;

    public Veiculo(string modelo) ⇒ Modelo = modelo;

    public virtual void Acelerar() ⇒ Veloc += 10;
}
```

```
class Automovel : Veiculo
{
    public int NumeroDePortas { get; set; }

    public Automovel(string modelo, int portas) :
        base(modelo) ⇒ NumeroDePortas = portas;

    public override void Acelerar() ⇒ Veloc += 20;
}
```

```
class Caminhao : Veiculo
{
    public int CapacCarga { get; set; }

    public Caminhao(string modelo, int capacCarga)
        : base(modelo) ⇒ CapacCarga = capacCarga;

    public override void Acelerar() ⇒ Veloc += 5;
}
```

Sobrescrita de Métodos

```
var veiculoGenerico1 = new Veiculo("Honda CG 160");
var carro1 = new Automovel("Volkswagen Polo", 4);
var caminhao1 = new Caminhao("Volvo FH 450", 38000);

veiculoGenerico1.Acelerar(); // Aumenta a velocidade em 10;
carro1.Acelerar(); // Aumenta a velocidade em 20;
caminhao1.Acelerar(); // Aumenta a velocidade em 5;

Console.WriteLine(veiculoGenerico1.Veloc); // Mostra 10
Console.WriteLine(carro1.Veloc); // Mostra 20
Console.WriteLine(caminhao1.Veloc); // Mostra 5
```

Referenciando a Classe Pai com a Palavra base

- Há situações onde o código da classe filha precisa acessar o método virtual original da classe pai;
- Nesses casos, basta usar a sintaxe base.MetodoDaClassePai().

```
class Automovel : Veiculo
{
    public int NumeroDePortas { get; set; }

    public Automovel(string modelo, int portas) :
        base(modelo) ⇒ NumeroDePortas = portas;

    public override void Acelerar() ⇒ Veloc += 20;

    // Método para acelerar forte - aumentará a velocidade em 30
    public void AcelerarForte()
    {
        base.Acelerar(); // Chama o método Acelerar da classe pai (+10)
        Acelerar(); // Chama o método Acelerar desta classe (+20)
    }
}
```

Exercício

- Crie uma classe base de nome Funcionario. A classe deve possuir:
  - Duas propriedades públicas: Nome (string) e Salario (double).
  - Um construtor para inicializar as propriedades.
  - Um método virtual chamado CalcularBonificacao() que retorne o valor de 10% do salário.
- Crie uma classe derivada de nome Gerente. A classe deve sobrescrever o método CalcularBonificacao() da classe base para que seja retornado uma bonificação maior, correspondente a 20% do salário.
- Crie um programa principal ilustrando o uso das classes e métodos.

Classes Abstratas

- Em algumas situações é possível que a classe pai, isoladamente, não faça sentido no mundo real.
- Por exemplo, ao definir as classes Pessoa, PessoaFisica e PessoaJuridica, não faz sentido permitir a criação de objetos da classe Pessoa, uma vez que no mundo real a pessoa é sempre física ou jurídica.
- Outro exemplo seria a definição da classe FiguraGeometrica e das classes filhas Circulo, Retangulo e Triangulo. Na prática, para fins de exibição, não faz sentido criar objetos genéricos do tipo FiguraGeometrica.
- Nessas situações, a classe pai pode ser definida como abstrata, enquanto as demais continuam sendo classes concretas convencionais.
- Classes abstratas NÃO podem ser instanciadas diretamente.

Exemplo de Classe Abstrata - Definição

```
abstract class Pessoa
{
    public string Nome { get; set; }
    public Pessoa(string nome) => Nome = nome;
}

class PessoaFisica : Pessoa
{
    public string Cpf { get; set; }
    public PessoaFisica(string nome, string cpf)
        :base(nome) => Cpf = cpf;
}

class PessoaJuridica : Pessoa
{
    public string Cnpj { get; set; }
    public PessoaJuridica(string nome, string cnpj)
        : base(nome) => Cnpj = cnpj;
}
```

Exemplo de Classe Abstrata – Programa Principal

```
// ERRO: Pessoa é uma classe abstrata e
// não pode ser instanciada diretamente.
var p1 = new Pessoa("Fulano");

// PessoaFisica e PessoaJuridica são classes
// concretas e podem ser instanciadas normalmente.
var pFisica = new PessoaFisica("Fulano", "123");
var pJuridica = new PessoaJuridica("Empresa A", "456");
```

Métodos e Propriedades Abstratos

- Em classes abstratas é comum haver a definição de métodos abstratos (e propriedades abstratas).
- Métodos abstratos não possuem um corpo, mas apenas uma assinatura (nome e parâmetros).
- Métodos abstratos atuam como um contrato que obriga a classe filha a fornecer sua própria implementação.
  - Na classe filha, o método precisará ser declarado com override.
- São definidos com a palavra-chave abstract.
  - `public abstract double CalcularArea();`
- Membros abstratos e virtuais não podem ser definidos como private.

Classes e Métodos Abstratos - Exemplo

```
abstract class FiguraGeometrica
{
    // Toda figura terá uma cor de fundo,
    // uma cor de borda e coordenadas de exibição.
    public Cor? CorDeFundo { get; set; }
    public Cor? CorDaBorda { get; set; }
    public int PosX { get; set; }
    public int PosY { get; set; }

    // Método concreto para mover qualquer figura.
    public void Mover(int x, int y)
    {
        PosX += x;
        PosY += y;
    }

    // Método abstrato. Todas as classes filhas
    // precisam implementa-lo com fórmula própria.
    public abstract double CalcularArea();

    // Propriedade abstrata. Deve ser implementada
    // nas classes filhas para que toda figura
    // tenha um nome.
    public abstract string Nome { get; }
}
```

```
class Circulo : FiguraGeometrica
{
    // Propriedade específica do círculo.
    public double Raio { get; set; }

    // Construtor
    public Circulo(double raio) => Raio = raio;

    // Implementação obrigatória, pois foram
    // declarados como abstratos na classe pai.
    public override string Nome => "Círculo";
    public override double CalcularArea()
        => Math.PI*Raio*Raio;
}
```

Definição das Classes Retangulo e Triangulo

```
class Retangulo : FiguraGeometrica
{
    // Propriedades específicas do retângulo.
    public double Largura { get; set; }
    public double Altura { get; set; }

    // Construtor
    public Retangulo(double larg, double alt)
    {
        Largura = larg;
        Altura = alt;
    }

    // Implementação obrigatória, pois foram
    // declarados como abstratos na classe pai.
    public override string Nome ⇒ "Retângulo";
    public override double CalcularArea()
        ⇒ Largura * Altura;
}
```

```
class Triangulo : FiguraGeometrica
{
    // Propriedades específicas do triângulo.
    public double Base { get; set; }
    public double Altura { get; set; }

    // Construtor
    public Triangulo(double @base, double alt)
    {
        Base = @base;
        Altura = alt;
    }

    // Implementação obrigatória, pois foram
    // declarados como abstratos na classe pai.
    public override string Nome ⇒ "Triângulo";
    public override double CalcularArea()
        ⇒ (Base * Altura) / 2;
}
```

Uso das Classes Circulo, Retangulo e Triangulo

```
// ERRO: Não podemos criar um objeto da classe FiguraGeometrica,
// pois ela foi definida como abstrata.
var figuraGenerica = new FiguraGeometrica(); // ERRO!

var fig1 = new Circulo(5);
var fig2 = new Retangulo(10, 5);
var fig3 = new Triangulo(10, 20);

Console.WriteLine(fig1.Nome); // Mostra 'Círculo'
Console.WriteLine(fig2.Nome); // Mostra 'Retângulo'
Console.WriteLine(fig3.Nome); // Mostra 'Triângulo'

Console.WriteLine(fig1.CalcularArea()); // Chama o método de cálculo de área do círculo
Console.WriteLine(fig2.CalcularArea()); // Chama o método de cálculo de área do retângulo
Console.WriteLine(fig3.CalcularArea()); // Chama o método de cálculo de área do triângulo
```

Métodos Abstratos vs Métodos Virtuais

- Ao definir um método como virtual, você está apenas permitindo que o método possa ser sobrescrito, se necessário, nas classes derivadas.
- O método virtual pode ter sua própria implementação na classe base.
- Por outro lado, métodos abstratos não podem ter um corpo (não podem ser implementados na classe base). Portanto, é obrigatória a sua implementação/sobrescrita nas classes derivadas.
- Métodos abstratos só podem ser declarados dentro de classes abstratas, enquanto métodos virtuais podem ser declarados também em classes concretas.

Exercício

1. Crie uma classe abstrata de nome Pagamento contendo uma propriedade para armazenar o valor original do pagamento. Deve haver um construtor para inicialização da propriedade e um método abstrato CalcularTotalAPagar(), com retorno em double, que obrigue as classes derivadas a definirem eventuais descontos ou acréscimos para cálculo do valor final a ser pago.
2. Crie uma classe derivada concreta de nome PagamentoPix. A classe deve implementar o método CalcularTotalAPagar() aplicando um desconto de 10% sobre o valor original do pagamento (multiplique o valor original por 0.9). O valor descontado deve ser retornado. Deve haver um construtor adequado.
3. Crie uma classe derivada concreta de nome PagamentoCredito que contenha uma propriedade própria de nome NumeroDeParcelas (int) e um construtor para inicialização das propriedades. A classe deve implementar o método CalcularTotalAPagar() acrescentando juros simples de 2% ao mês ao valor original, conforme número de parcelas (valor total = valor original * (1 + (0.02 * número de parcelas)).
4. Crie um programa principal ilustrando o uso das classes e dos métodos.

Polimorfismo e Ligação Dinâmica

- Polimorfismo significa "muitas formas" (do grego: *poly* = muitas, *morph* = formas)
- Um objeto de uma classe filha, como Circulo, pode ser tratado como um objeto do tipo Circulo de fato, mas também como um objeto do tipo FiguraGeometrica, dependendo da situação.
- Nesse contexto, o objeto é tratado na prática de duas formas diferentes, caracterizando o polimorfismo (veja exemplo no próximo slide).

Polimorfismo e Ligação Dinâmica

```
// Criação de objetos das classes especializadas.
var circulo1 = new Circulo(10);
var retangulo1 = new Retangulo(20, 30);
var triangulo1 = new Triangulo(5, 10);

// Todos os objetos especializados são armazenados em um
// array de objetos genéricos. Aqui, os objetos são todos tratados
// pelo compilador do C# como do tipo FiguraGeometrica.
FiguraGeometrica[] minhasFiguras = [circulo1, retangulo1, triangulo1];

// Em um único laço, calculamos e apresentamos a área de todas as
// figuras geométricas, independentemente do seu tipo,
// chamando a mesma função CalcularArea()
foreach (var figura in minhasFiguras)
{
    // O compilador trata a variável figura como do tipo FiguraGeometrica,
    // mas o tipo real do objeto em memória (Circiulo, Retangulo ou Triangulo)
    // será verificado em tempo de execução e resultará na chamada correta
    // do método CalcularArea() conforme o tipo específico da figura.
    // Isto é chamado de 'Ligação Dinâmica' ou 'Ligação Tardia'.
    // Neste ponto, os objetos são tratatos, em tempo de execução, conforme
    // o tipo específico de cada um.
    Console.WriteLine(figura.CalcularArea());
}
```

Ligação Dinâmica:
A decisão pela versão do método a ser chamada ocorre em tempo de execução, conforme o tipo real do objeto naquele momento.

Ligação Estática:
Em uma chamada convencional, o método correto é escolhido em tempo de compilação.

Polimorfismo e Ligação Dinâmica

```
// Criação de objetos das classes especializadas.
var moto1 = new Veiculo("Honda CG 160");
var carro1 = new Automovel("Volkswagen Polo", 4);
var caminhao1 = new Caminhao("Volvo FH 450", 38000);

// Todos os objetos especializados são armazenados em um
// array de objetos genéricos (Veiculo).
Veiculo[] arrayVeiculos = [moto1, carro1, caminhao1];

// Em um único laço, todos os veículos do array serão acelerados,
// independentemente do seu tipo (automóvel, caminhão etc.), mas
// a aceleração ocorrerá de forma diferente em cada tipo de veiculo,
// conforme cada método sobrescrito na classe especializada.
foreach (var veiculo in arrayVeiculos)
    veiculo.Acelerar();
```

Este exemplo é similar ao anterior e também ilustra o polimorfismo e a ligação dinâmica.

Polimorfismo e Ligação Dinâmica - Vantagens

- Tornam o código mais flexível e extensível
  - Se for necessário implementar mais uma classe especializada de veículo, o laço foreach que percorre os elementos do array no exemplo anterior continuará funcionando sem alterações.
  - Se um novo tipo de figura geométrica for criada, o laço que calcula as áreas de todas as figuras não precisará ser alterado.
- Permite um código mais simples e desacoplado. Sem polimorfismo, no exemplo anterior, teríamos um longo if-else:

```
if (veiculo is Automovel) { "acelerarAutomóvel" }
else if (veiculo is Caminhao) { "acelerarCaminhao" }
else { ... }
```

Exercício

- Defina uma classe abstrata para modelar animais de uma forma geral. A classe deve ter uma propriedade pública concreta para armazenar o nome do animal e um método abstrato EmitirSom().
- Defina duas classes especializadas para modelar, respectivamente, cães e gatos.
  - O construtor deve ter um parâmetro para inicialização do nome.
  - Cada classe deve ter sua própria definição do método EmitirSom() que apresente na tela uma mensagem correspondente ao som do respectivo animal.
- Crie um "programa principal" ilustrando o uso das classes. O programa deve criar um array de objetos do tipo Animal e percorrer o array utilizando um laço foreach para apresentação do nome e emissão do respectivo som de cada objeto do array.

Interfaces

- Em POO, uma interface é uma especificação que atua como um contrato que estabelece uma série de funcionalidades que as classes devem ter, mas sem necessariamente* prover uma implementação para elas.
- Permitem que objetos de naturezas distintas (como Veiculos e Pessoas) possam ter alguns comportamentos em comum (ex. localização)
- Quando uma classe implementa a interface, ela está "assinando o contrato", e portanto se torna obrigada a oferecer aquelas funcionalidades.
- A interface define O QUÊ a classe deve fazer, mas não COMO ela deve fazer.
- A interface contem as assinaturas dos métodos e propriedades que devem ser implementados pela classe, mas não o código em si.
- A interface não tem campos de dados (atributos) e todas as suas definições (métodos, propriedades, etc.) são implicitamente public.
- Por convenção, nomes de interfaces em C# deve começar com o `I` maiúsculo (exemplo: IMinhaInterface).

*Interfaces em C# (v8+) podem ter uma implementação padrão de métodos, mas esse assunto não será abordado neste material.

Criando uma interface

Interfaces são criadas com a palavra-chave interface

```
interface IApresentavel
{
    // Exige que a classe tenha uma propriedade
    // que retorne o nome do objeto.
    string Nome { get; }

    // Exige que a classe tenha um método sem parâmetros
    // e sem retorno para apresentação dos objetos.
    // OBS: Não diz como a apresentação tem
    // que ser (por mensagem de texto, arquivo, tela etc.),
    // mas apenas que tem que ter uma forma de apresentação.
    void Mostrar();
};
```

Implementando uma interface - Exemplo 1

- Para que uma classe implemente a interface, deve-se utilizar o operador dois-pontos de maneira similar àquela utilizada na criação de uma classe derivada.
- A classe deve fornecer uma implementação para todas as definições da interface.

```
// Define uma classe 'assinando' o contrato da interface
class Quadrado : IApresentavel
{
    // Propriedade específica do círculo.
    public double Lado { get; set; }

    // Construtor
    public Quadrado(double lado) => Lado = lado;

    // Propriedade EXIGIDA pela interface IApresentavel
    public String Nome => "Quadrado";

    // Método EXIGIDO pela interface IApresentavel
    public void Mostrar() {
        Console.WriteLine($"Sou um quadrado de lado {Lado}");
    }
}
```

Implementando uma interface - Exemplo 2

```
class Motocicleta : IApresentavel
{
    public int Cilindradas { get; set; }

    public Motocicleta(int cilindradas) => Cilindradas = cilindradas;

    // Propriedade exigida pela interface IApresentavel
    public string Nome => "Motocicleta";

    // Método exibido pela interface IApresentavel
    public void Mostrar() => Console.WriteLine($"Moto de {Cilindradas} cilindradas");
}
```

Interfaces, Polimorfismo e Ligação Dinâmica

```
// Um array do tipo IApresentavel poderá armazenar objetos de qualquer
// classe, desde que a classe implemente a interface IApresentavel
IApresentavel[] objetosApresentaveis = [quad1, quad2, moto1, moto2];
foreach (var objeto in objetosApresentaveis)
{
    // Como o objeto específico será de uma classe que implementa
    // a interface, temos a certeza de que a propriedade Nome
    // e o método Mostrar() estão disponíveis.
    Console.WriteLine(objeto.Nome);
    objeto.Mostrar();
}
```

Declaração de Variáveis utilizando Interfaces

- Variáveis e parâmetros podem ser declarados utilizando nomes de interfaces.
- Nesse caso, a variável poderá receber objetos de qualquer classe que implementa a interface.

```
class MinhaClasse
{
    // Na chamada deste método poderá ser passado um objeto
    // de qualquer classe, desde que a classe implemente
    // a interface IApresentavel
    public void MeuMetodo(IApresentavel objApresentavel)
    {
        objApresentavel.Mostrar();
    }
}
```

Classe Derivada com Implementação de Interface

- É possível definir uma classe derivada que ao mesmo tempo implementa interfaces

```
abstract class Veiculo
{
    public string Modelo { get; }
    public int Veloc { get; set; } = 0;

    public Veiculo(string modelo)
    {
        Modelo = modelo;
    }

    public abstract void Acelerar();
}
```

```
// Classe concreta derivada da classe abstrata Veiculo
// implementando a interface IApresentavel
class Motocicleta : Veiculo, IApresentavel
{
    public int Cilindradas { get; set; }

    public Motocicleta(String modelo, int cilindradas)
        : base(modelo) => Cilindradas = cilindradas;

    // Método Acelerar EXIGIDO pela classe abstrata Veiculo
    public override void Acelerar() => Veloc = Veloc + 10;

    // Propriedade EXIGIDO pela interface IApresentavel
    public string Nome => "Motocicleta";

    // Método EXIGIDO pela interface IApresentavel
    public void Mostrar() => Console.WriteLine(
        $"Moto de {Cilindradas} cilindradas");
}
```

Herança Múltipla

- A linguagem C# (assim como Java) não suporta a definição de uma classe derivada que herda de múltiplas classes simultaneamente.
- Entretanto, uma classe pode implementar múltiplas interfaces:
  - class MinhaClasse : ClasseBase, Interface1, Interface2, Interface3 { ... }

Exercício

- Defina uma interface ILocalizavel que contenha a definição do método CoordenadaGPS, sem parâmetro, com retorno em string.
- Defina uma classe abstrata Pessoa que contenha uma propriedade Nome e um construtor para sua inicialização. A classe deve possuir também uma definição de método abstrato CalcularRendimentoAnual() com retorno em double.
- Defina uma classe concreta PessoaFisica, que herde de Pessoa e implemente a interface ILocalizavel. A classe deve possuir uma propriedade específica CPF e um construtor para correta inicialização. Para calcular o rendimento anual, considere que toda pessoa física possui rendimento mensal de 5 salários mínimos. O método de cálculo da localização deve retornar a coordenada fictícia "lat -18.9, long -48.2".
- Defina uma classe concreta Automovel que contenha as propriedades placa (string), modelo (string) e ano (int) e um construtor para inicialização das propriedades. A classe deve implementar a interface ILocalizavel. Considere que todos os veículos estejam localizados em "lat -15.7, long -47.8".
- Crie um programa principal para ilustrar o uso das classes, métodos e interface. O programa deve ilustrar o conceito de polimorfismo e ligação dinâmica.

Referências

- Joseph Albahari. **C# 12 in a Nutshell**: The Definitive Reference. 1ª ed., 2023.
- Mark J. Price. **C# 9 and .NET 5 – Modern Cross-Platform Development**: Build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code, 5ª ed., 2020.
- learn.microsoft.com/en-us/dotnet/csharp/fundamentals