

Orientação a objetos

Prof. Ricardo Frohlich da Silva

Abstração

- A abstração é um dos princípios fundamentais da programação orientada a objetos (POO) e desempenha um papel crucial no desenvolvimento de software em C# e em muitas outras linguagens.
- Em termos simples, abstração é o processo de simplificar um conceito complexo, permitindo que você se concentre nos detalhes mais relevantes enquanto oculta os detalhes mais complexos e menos importantes.
- Em C#, a abstração é alcançada através de classes abstratas e interfaces.

• Material de apoio: <https://learn.microsoft.com/pt-br/dotnet/csharp/fundamentals/tutorials/inheritance>

• https://www.youtube.com/watch?v=E8WJ_z-osqE

Herança

- *A abstração ajuda a diminuir a complexidade.*
- *Encapsulamento ajuda a gerenciar essa complexidade, ocultando a visão dentro de nossa abstrações.*
- *A modularidade também ajuda, dando-nos uma maneira de agrupar logicamente abstrações relacionadas.*
- *Um conjunto de abstrações, muitas vezes forma uma hierarquia, e identificando essas hierarquias no nosso projeto, simplifica grandemente o nossa compreensão do problema.*
- *Material de apoio: <https://learn.microsoft.com/pt-br/dotnet/csharp/fundamentals/tutorials/inheritance>*

Herança

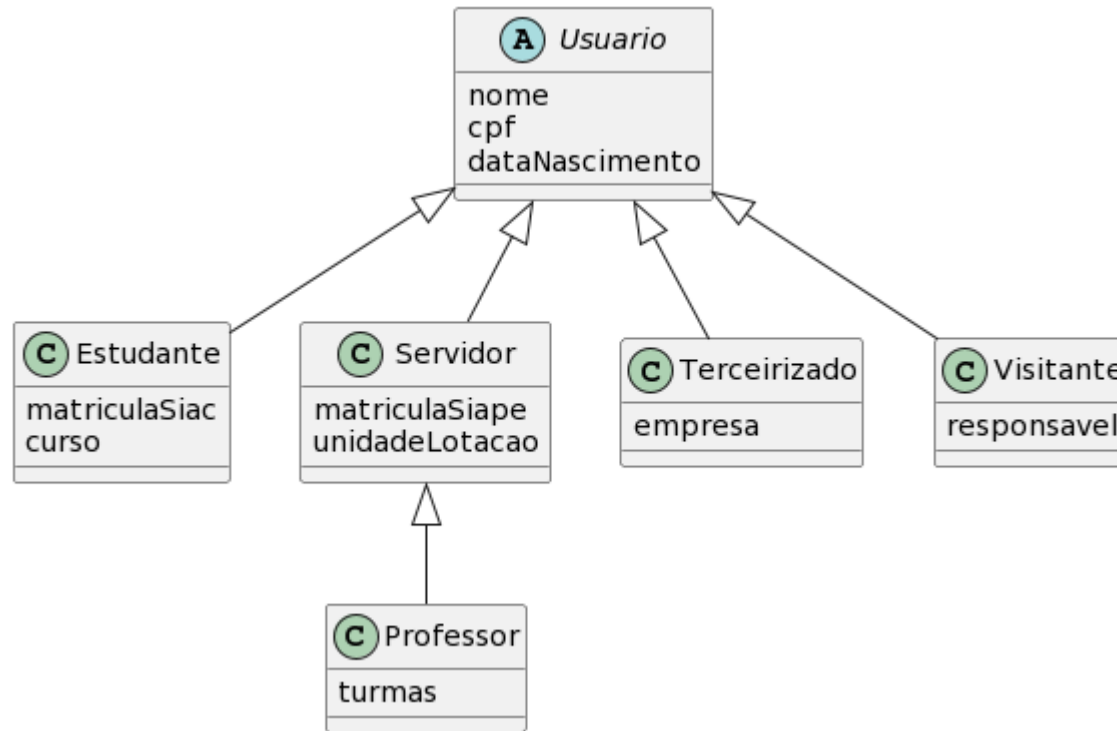
- Herança é o mecanismo para expressar a similaridade entre Classes, simplificando a definição de classes iguais que já foram definidas.
- O que um leão, um tigre, um gato, um lobo e um dálmatas têm em comum?
- Como eles são relacionados?

Abstração

- Uma classe abstrata em C# é uma classe que não pode ser instanciada diretamente, mas serve como uma estrutura base para outras classes derivadas.
- Ela pode conter membros abstratos (métodos e propriedades) que as classes derivadas são obrigadas a implementar.

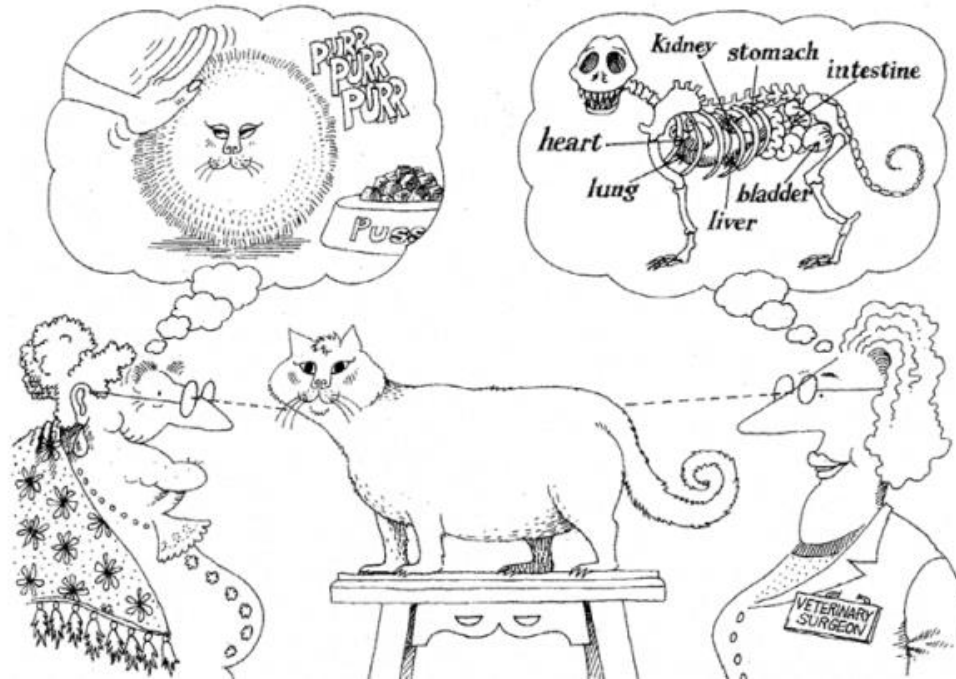
Classes abstratas

- Classes abstratas geralmente representam um conjunto de dados ou funcionalidades incompletas, que precisam ser completados pelas subclasses. Exemplo:



Abstração

- Foca a característica essencial de alguns objetos relativo a perspectiva do visualizador



Métodos abstratos

- Um método abstrato não possui implementação (corpo)
- Apenas classes abstratas podem possuir métodos abstratos
- A implementação dos métodos abstratos deve ser fornecida pelas subclasses concretas
- Exemplo:

```
public abstract void meuMetodo();
```


Construindo uma classe abstrata

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

    public abstract void EmitirSom();
}
```

Implementando uma classe

```
public class Cachorro : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine("Au! Au!");
    }
}
```

Instanciando

```
static void Main(string[] args)
{
    Cachorro c = new Cachorro();
    c.Nome = "Hugo";
    c.EmitirSom();
}
```

Jogo rápido

- Faça a implementação de uma classe Gato

Jogo rápido

- Faça a implementação de uma classe Gato

```
internal class Gato : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine("Miau!!");
    }
}
```

Jogo rápido

- Adicione na classe gato um atributo chamado Vidas
- Crie um método ronronar e apresente uma mensagem na tela

Um pouco mais sobre as possibilidades

- Além disso, cada implementação concreta de uma classe abstrata pode possuir métodos e atributos que são específicos desta implementação:

```
internal class Gato : Animal
{
    public int Vidas { get; set; }

    public override void EmitirSom()
    {
        Console.WriteLine("Miau!!");
    }

    public void Ronronar()
    {
        Console.WriteLine("Gato ronronando...");
    }
}
```

Jogo rápido

- Criar um método se Movimentar na classe Animal

Jogo rápido

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

    public abstract void EmitirSom();

    public void Movimentar()
    {
        Console.WriteLine("O animal está se movimentando!");
    }
}
```

- E desta forma, agora os objetos da classe Cachorro e Gato poderão chamar o método Movimentar, ou seja:
- O método Movimentar() é herdado por suas subclasses, assim como a propriedade Nome!

Um pouco mais sobre as possibilidades

- Além disso, cada implementação concreta de uma classe abstrata pode possuir métodos e atributos que são específicos desta implementação:

```
static void Main(string[] args)
{
    Cachorro c = new Cachorro();
    c.Nome = "Hugo";
    c.EmitirSom();

    Gato g = new Gato();
    g.Nome = "Garfield";
    g.EmitirSom();
    g.Ronronar();
}
```

Classe abstrata com construtor

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

    public Animal(string nome)
    {
        Nome = nome;
    }

    public abstract void EmitirSom();
}
```

Classe abstrata com construtor

```
public class Cachorro : Animal
{
    public Cachorro(string nome) : base(nome)
    {
    }

    public override void EmitirSom()
    {
        Console.WriteLine("O cachorro late: Au Au!");
    }
}
```

Classe abstrata com construtor

```
static void Main(string[] args)
{
    Cachorro c = new Cachorro("Hugo");
    Console.WriteLine("Cachorro chamado: "+c.Nome);
    c.EmitirSom();
}
```

Classe abstrata com construtor

- E se eu quiser alguns atributos a mais no construtor, como por exemplo, na classe Gato que existe Vidas?

Classe abstrata com construtor

```
internal class Gato : Animal
{
    public int Vidas { get; set; }
    public Gato(string nome, int vidas) : base(nome)
    {
        Vidas = vidas;
    }

    public override void EmitirSom()
    {
        Console.WriteLine("Miau!!");
    }

    public void Ronronar()
    {
        Console.WriteLine("Gato ronronando...");
    }
}
```

Classe abstrata com construtor

```
internal class Program
{
    static void Main(string[] args)
    {
        Cachorro c = new Cachorro("Hugo");
        Console.WriteLine("Cachorro chamado: "+c.Nome);
        c.EmitirSom();
        c.Movimentar();

        Gato g = new Gato("Garfield", 7);
        Console.WriteLine("Gato chamado: "+g.Nome);
        Console.WriteLine("E possui "+g.Vidas+" vidas!");
        g.EmitirSom();
        g.Ronronar();
        g.Movimentar();
    }
}
```


Interfaces

- Uma interface define um contrato que as classes devem cumprir, especificando um conjunto de métodos e propriedades que as classes que a implementam devem fornecer.
- As interfaces permitem uma abstração mais completa, onde várias classes podem implementar a mesma interface sem a necessidade de herança direta.

```
public interface IVeiculo
{
    void Acelerar();
    void Parar();
}
```

Interfaces

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

    public void Parar()
    {
        Console.WriteLine("O carro parou.");
    }
}
```

Interfaces

- Crie um método dentro da interface que exiba alguma mensagem

```
public interface IVeiculo
{
    void Acelerar();
    void Parar();
    public void ExibirMensagem()
    {
        Console.WriteLine("Estou na interface IVeiculo!");
    }
}
```

Interfaces

- Instancie e chame na Main:

```
static void Main(string[] args)
{
    Carro c = new Carro();
    c.Acelerar();
    c.Parar();
    c.ExibirMensagem();
}
```

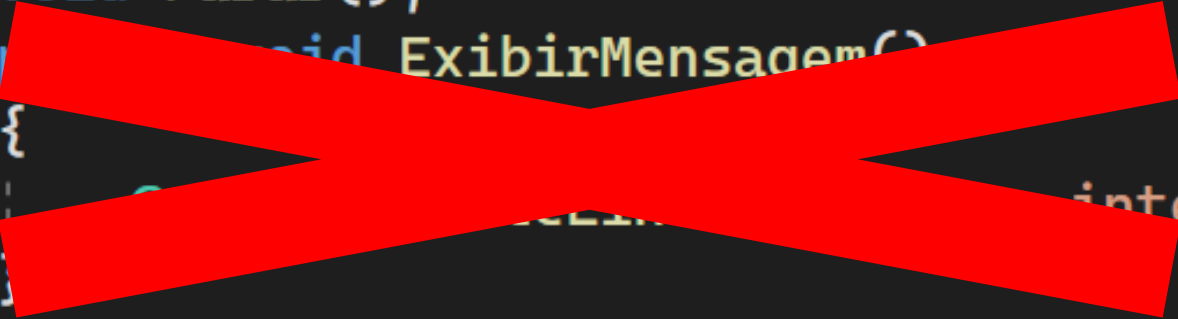
Interfaces

- Ocorreu o erro pois ele procura o método na classe concreta, ou seja, na classe Carro e não encontra!
- Mas, aí podemos implementar ela na classe Carro!

Interfaces

```
public interface IVeiculo
{
    void Acelerar();
    void Parar();
    void ExibirMensagem();
}

// interface IVeiculo!");
```



- Não teremos **NUNCA** implementação concreta em uma Interfaces

Interfaces

```
public class Carro : IVeiculo
{
    public string Nome { get; set; }
    public void Acelerar()
    {
        Console.WriteLine("O carro está acelerando.");
    }

    public void Parar()
    {
        Console.WriteLine("O carro parou.");
    }

    public void ExibirMensagem()
    {
        Console.WriteLine("Carro: "+Nome);
    }
}
```

```
static void Main(string[] args)
{
    Carro c = new Carro();
    c.Nome = "Gol";
    c.Acelerar();
    c.Parar();
    c.ExibirMensagem();
}
```

Interfaces

```
public class Bicicleta : IVeiculo
{
    public void Acelerar()
    {
        Console.WriteLine("A bicicleta está acelerando.");
    }

    public void Parar()
    {
        Console.WriteLine("A bicicleta parou.");
    }
}
```


Interfaces

```
static void Main(string[] args)
{
    Carro c = new Carro();
    c.Nome = "Gol";
    c.Acelerar();
    c.Parar();
    c.ExibirMensagem();

    Bicicleta b = new Bicicleta();
    b.Acelerar();
    b.Parar();
}
```

Exercícios

- 1 - Criar uma classe Pessoa que contém as propriedades Nome e Idade. Criar duas classes derivadas, Aluno e Professor, que herdam de Pessoa.
- A classe Aluno deve ter uma propriedade adicional Matricula, enquanto a classe Professor deve ter uma propriedade adicional Disciplina.
- Criar um método abstrato Apresentar na classe Pessoa.
- Em seguida, criar um método Apresentar na classe Aluno que imprime o nome, a idade e a matrícula do aluno, e um método Apresentar na classe Professor que imprime o nome, a idade e a disciplina do professor.

Exercícios

- 2 - Criar uma classe Veiculo que contém as propriedades Marca e Modelo. Criar um método abstrato chamado Dirigir. Criar duas classes derivadas, Carro e Moto, que herdam de Veiculo.
- A classe Carro deve ter uma propriedade adicional QuantidadeDePortas, enquanto a classe Moto deve ter uma propriedade adicional Cilindrada.
- Criar um método abstrato Dirigir na classe Veiculo.
- Em seguida, criar um método Dirigir na classe Carro que imprime "Dirigindo o <marca> <modelo> com <quantidadeDePortas> portas" e um método Dirigir na classe Moto que imprime "Dirigindo a <marca> <modelo> com <cilindrada> cilindradas"

Exercícios

- 3 - Crie uma interface chamada "IFormaGeometrica" com dois métodos abstratos: "CalcularArea()" e "CalcularPerimetro()".
- Em seguida, crie duas classes que implementam essa interface: "Retangulo" e "Circulo".
- Implemente os métodos "CalcularArea()" e "CalcularPerimetro()" para cada uma dessas classes, de forma que o "Retangulo" calcule a área e o perímetro de um retângulo e o "Circulo" calcule a área e o perímetro de um círculo.

Exercícios

- 4 - Crie uma classe abstrata "ContaBancaria" com propriedades para "Saldo" e métodos abstratos "Depositar" e "Sacar". Crie classes derivadas, como "ContaCorrente" e "ContaPoupanca", que implementam os métodos de depósito e saque de acordo com as regras de cada tipo de conta.
- 5 - Crie uma classe abstrata "Produto" com propriedades para "Nome", "Preço" e um método abstrato "CalcularDesconto". Crie classes derivadas para diferentes tipos de produtos, como "Livro" e "Eletrônico", que implementam o método "CalcularDesconto" de acordo com as regras específicas de desconto para cada tipo de produto.
 - Livro – 5% de desconto
 - Eletrônico - 12.5% de desconto

Materiais extra para estudo

- <https://rodrigorgs.github.io/aulas/poo/aula-heranca-parte1.html#1>
- <https://rodrigorgs.github.io/aulas/poo/aula-heranca-parte2.html#1>
- <https://rodrigorgs.github.io/aulas/poo/aula-heranca-parte3.html#1>
- <https://rodrigorgs.github.io/aulas/poo/aula-heranca-parte4>