



Fundamentos de C#
Polimorfismo


Softblue
cursos online

Tópicos Abordados

- Polimorfismo
 - Polimorfismo com classes
 - Polimorfismo com interfaces
- Sobrescrita de métodos
 - Modificadores *virtual* e *override*
- Casting
 - Operadores *as* e *is*
- Classes e métodos abstratos
- O modificador *sealed*
- Implementação explícita de interfaces

Polimorfismo

- É um dos pilares da orientação a objetos



Polimorfismo

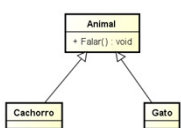


- O polimorfismo possibilita tratar vários tipos de dados através de forma homogênea
 - A referência a um objeto é feita através de um tipo mais genérico
 - No momento da execução, o que será executado pode variar de acordo com o objeto sendo referenciado

Herança e Implementação de Métodos



- Uma classe que herda de outra ou implementa uma interface tem acesso aos métodos/properties do tipo mais genérico



```
Cachorro c = new Cachorro();  
c.Falar();  
  
Gato g = new Gato();  
g.Falar();
```

Oi

Oi

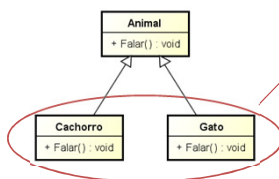
```
class Animal  
{  
    public void Falar()  
    {  
        Console.WriteLine("Oi");  
    }  
}
```

Como fazer para cada objeto ter seu próprio método *Falar()*?

Sobrescrita de Métodos



- Sobrescrever significa alterar a implementação de um método que foi declarado em um tipo mais genérico



As subclasses implementam suas próprias versões de *Falar()*

Modificadores *virtual* e *override*

• O processo de sobrescrita em C# é feito utilizando os modificadores *virtual* e *override*

- Podem ser usados em métodos ou properties

```

class Animal
{
    public virtual void Falar()
    {
        Console.WriteLine("Oi");
    }
}

class Cachorro : Animal
{
    public override void Falar()
    {
        Console.WriteLine("Au-Au");
    }
}

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

Indica que o método pode ser sobrescrito

Sobrescreve o método

Modificadores *virtual* e *override*

• Quando o método é sobrescrito, a nova implementação é chamada

```

Cachorro c = new Cachorro();
c.Falar();

Gato g = new Gato();
g.Falar();
    
```

Au-Au

Miau

Dependendo do tipo do objeto, um *Falar()* diferente é executado

Polimorfismo na Prática

• Quando uma classe herda de outra ou implementa uma interface, ela é do tipo da superclasse ou interface

```

Cachorro c = new Cachorro();
Animal a = new Cachorro();
    
```

OK!

OK!

Cachorro é um Animal

Uma variável de um tipo mais genérico referencia um objeto de tipo mais específico

Polimorfismo

Polimorfismo na Prática

Animal a1 = new Cachorro();
a1.Falar(); Au-Au

Animal a2 = new Gato();
a2.Falar(); Miau

void EmitirSom(Animal a)
{
 a.Falar();
}

Cachorro cachorro = new Cachorro();
EmitirSom(cachorro); Au-Au

Gato gato = new Gato();
EmitirSom(gato); Miau

Vaca vaca = new Vaca();
EmitirSom(vaca); Mu

O método a ser chamado depende do objeto referenciado na memória

O método recebe um animal qualquer como parâmetro

EmitirSom() está preparado para fazer qualquer animal falar

Polimorfismo na Prática

- Ao referenciar um objeto por um tipo mais genérico, não temos acesso aos elementos declarados no tipo mais específico

```

classDiagram
    class Animal {
        +Falar() void
    }
    class Cachorro {
        +Falar() void
        +Morder() void
    }
    class Gato {
        +Falar() void
    }
    Animal <|-- Cachorro
    Animal <|-- Gato
    
```

Cachorro c = new Cachorro();
c.Morder(); OK!

Animal a = new Cachorro();
a.Morder(); ERRO!

O tipo Animal não tem o método Morder()

Casting

Animal a = new Cachorro();
a.Morder();

Animal não tem acesso ao método Morder()

Na memória, a referencia um Cachorro

Cachorro tem o método Morder()

A solução é fazer um casting da referência

Cachorro c = (Cachorro) a;
c.Morder();

O casting transforma o tipo da variável que referencia o objeto

O casting jamais transforma o objeto

Casting

Softblue

- O casting pode ocasionar erros que aparecem apenas na execução

```
Animal a = new Cachorro();
Gato g = (Gato) a;
```

Não gera erro de compilação

Erro de execução: uma variável do tipo *Gato* não pode referenciar um objeto *Cachorro*

Casting

Softblue

- Antes de fazer o casting, precisamos nos certificar se ele realmente pode ser feito
- Em C#, isto pode ser feito através dos operadores **as** e **is**

```
Animal a = new Cachorro();
Cachorro c = a as Cachorro;
if (c != null)
{
    c.Morder();
}
```

Converte *a* para o tipo *Cachorro*

Se a conversão não for possível, retorna *null*

```
Animal a = new Cachorro();
if (a is Cachorro)
{
    Cachorro c = (Cachorro) a;
    c.Morder();
}
```

Checka o tipo fornecido e retorna um booleano

Fazer o casting é seguro

Classes Abstratas

Softblue

- Em algumas situações, determinadas classes existem apenas para fornecerem funcionalidades comuns às subclasses
 - Não existe sentido em termos uma instância direta da classe

```
Animal a = new Animal();
```

```
Cachorro c = new Cachorro();
```

```
Gato g = new Gato();
```

Representa um cachorro

Representa um gato

Representa o quê?

```

classDiagram
    Animal <|-- Cachorro
    Animal <|-- Gato
    
```

Classes Abstratas

• O modificador **abstract** é usado na classe para dizer que ela é abstrata

- Uma classe abstrata não pode ter instâncias diretas

```
abstract class Animal
{
}
```

```
Gato g = new Gato();
```

É permitido instanciar a subclasse

```
Animal a = new Animal();
```

Erro de compilação

```
Animal a = new Gato();
```

É permitido referenciar um objeto como *Animal*
Na memória, o objeto é do tipo **Gato**

Métodos Abstratos

• Seguem a mesma ideia das classes abstratas

- Usados quando não tem sentido ter uma implementação do método na superclasse

```
Cachorro c = new Cachorro();
c.Falar();
```

```
Gato g = new Gato();
g.Falar();
```

```
Animal a = new Animal();
a.Falar();
```

Au-Au

Miau

???

O método *Falar()* de *Animal* não tem sentido

Métodos Abstratos

• Nestes casos um método pode ser marcado com o modificador **abstract**

- Métodos abstratos não têm implementação

```
abstract class Animal
{
    public abstract void Falar();
}
```

Método abstrato, sem implementação

Regras para Métodos Abstratos

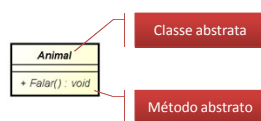


- Um método abstrato só pode ser declarado em uma classe abstrata
- Classes abstratas podem ter métodos abstratos e não abstratos
- Quando uma classe não abstrata herda de uma superclasse abstrata, ela é obrigada a sobrescrever todos os métodos abstratos declarados pela superclasse
 - Neste caso, é a subclasse quem vai fornecer a implementação dos métodos
- Todos os métodos abstratos são virtuais

UML e Classes/Métodos Abstratos



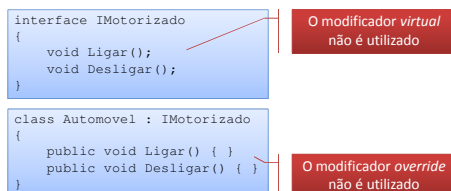
- Em UML, classes abstratas e métodos abstratos têm o seu nome definido em *itálico*



Polimorfismo com Interfaces



- O mesmo polimorfismo usado na herança pode ser aplicado também com interfaces



O modificador *virtual* não é utilizado

O modificador *override* não é utilizado

Polimorfismo com Interfaces

Softblue

```
IMotorizado m = new Automovel();
```

Referência usando um tipo mais genérico

```
Automovel a = m as Automovel;
if (a != null)
{
    //...
}
```

Operador as

```
if (a is Automovel)
{
    Automovel a = (Automovel)m;
    //...
}
```

Operador is

Classes Abstratas X Interfaces

Softblue

```
abstract class Motorizado
{
    public abstract void Ligar();
    public abstract void Desligar();
}
```

×

```
interface IMotorizado
{
    void Ligar();
    void Desligar();
}
```

- **Membros**
 - Classes abstratas podem definir fields, construtores, métodos e properties
 - Interfaces podem definir apenas métodos e properties
- **Herança/Implementação**
 - Uma classe só pode ter uma superclasse direta
 - Uma classe pode implementar várias interfaces
- **Hierarquia**
 - Uma classe abstrata só pode participar de uma hierarquia de classes
 - Uma interface pode ser usada de forma independente da hierarquia de classes

Invocando Métodos da Superclasse

Softblue

- Métodos da superclasse podem ser chamados pela subclasse usando a palavra-chave **base**
 - *base* referencia a superclasse

```
class Cachorro : Animal
{
    public override void Falar()
    {
        base.Falar();
    }
}
```

Chama o método da superclasse que está sendo sobrescrito

Pode ser usado para chamar qualquer método

Modificador *sealed*



- Classes definidas como *sealed* não podem ser herdadas
- Métodos/properties definidos como *sealed* não podem ser sobrescritos por subclasses

```
class Cachorro : Animal
{
    public override sealed void Falar()
    {
        Console.WriteLine("Au-Au");
    }
}
```

```
class Viralata : Cachorro
{
    public override void Falar() { }
```

A sobrescrita não é possível

Implementação Explícita de Interfaces



- A implementação explícita de elementos de uma interface esconde o elemento do objeto da classe

```
interface IMotorizado
{
    void Ligar();
}
```

```
class Automovel : IMotorizado
{
    public void Ligar() { }
```

```
class Automovel : IMotorizado
{
    void IMotorizado.Ligar() { }
```

Implementação explícita

```
Automovel a = new Automovel();
a.Ligar();
```

```
Automovel a = new Automovel();
a.Ligar();
```

Ligar() não é acessível

```
IMotorizado m = new Automovel();
m.Ligar();
```

Implementação Explícita de Interfaces



- Esta técnica pode ser usada para resolver ambiguidades no nome de métodos/properties


```
interface IMotorizado
{
    void Ligar();
}
```

```
interface ITelefone
{
    void Ligar();
}
```

```
class MyClass : IMotorizado, ITelefone
{
    public void Ligar() { }
```

A implementação de Ligar() é a mesma

Implementação Explícita de Interfaces



```
class MyClass : IMotorizado, ITelefone
{
    void IMotorizado.Ligar()
    {
        Console.WriteLine("Ligando motor");
    }

    void ITelefone.Ligar()
    {
        Console.WriteLine("Discando número");
    }
}
```

Implementações diferentes para cada *Ligar()*

MyClass m = new MyClass();

m.Ligar();

Não compila

((IMotorizado)m).Ligar();

"Ligando motor"

((ITelefone)m).Ligar();

"Discando número"



Softblue
