

# Fundamentos de C#

## Herança



---

---

---

---

---

---

---

### Tópicos Abordados



- Herança
  - Simples X Múltipla
- Herança em C#
- Os modificadores *protected* e *sealed*
- Classe *System.Object*
- Hierarquia nos tipos de dados
- Construtores e a herança
- Interfaces
  - Relação entre classes e interfaces
  - Implementação e herança
- Associação
- Exemplo de interface: *ICloneable*

---

---

---

---

---

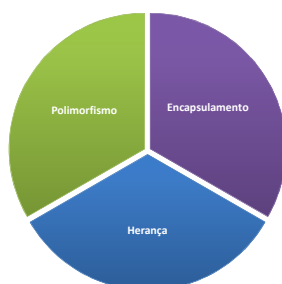
---

---

### Herança



- É um dos pilares da orientação a objetos



---

---

---

---

---

---

---

## Herança



- A herança é um relacionamento que pode existir entre classes na orientação a objetos
- Quando uma classe **B** herda de **A**
  - B herda as funcionalidades de A
  - B pode estender as funcionalidades de A
  - B passa a ser do mesmo tipo de A
- A herança cria um relacionamento entre classes do tipo “é-um”
- Estruturas (*structs*) não podem participar de uma relação de herança

---

---

---

---

---

---

---

## Terminologia



- Classes envolvidas em uma relação de herança podem ser chamadas de várias formas
- Suponha que **B** herde de **A**
  - **A**
    - é a superclasse de B
    - é a classe-mãe de B
    - é a classe-pai de B
    - é a classe base de B
  - **B**
    - é a subclasse de A
    - é a classe-filha de A
    - é a classe derivada de A
    - herda de A
    - estende A

---

---

---

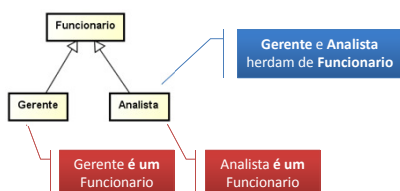
---

---

---

---

## Exemplo de Herança e UML



---

---

---

---

---

---

---

## Herança Simples X Múltipla

- Na herança simples, a classe tem apenas uma superclasse direta

```

graph BT
    AnalistaSenior --> Analista
    Analista --> Funcionario
        
```

- Na herança múltipla, a classe pode ter mais de uma superclasse direta

```

graph BT
    Analista --> Voluntario
    Analista --> Funcionario
        
```

C# não suporta herança múltipla

---

---

---

---

---

---

---

---

## Herança em C#

- A herança é determinada através da seguinte notação

```

class Subclasse : Superclasse
{
}

class Funcionario : Gerente
{
}
        
```

Funcionario estende Gerente

---

---

---

---

---

---

---

---

## Herança de Funcionalidades

- Uma subclasse herda as funcionalidades da sua superclasse

```

class Automovel
{
    public int Velocidade { get; set; }

    public void Acelerar()
    {
        Velocidade++;
    }

    public void Frear()
    {
        Velocidade = 0;
    }
}
        
```

```

class Onibus : Automovel
{
}

Onibus o = new Onibus();
o.Acelerar();
o.Frear();
o.Velocidade = 40;
        
```

O objeto Onibus tem acesso aos elementos de Automovel

---

---

---

---

---

---

---

---

## Extensão de Funcionalidades



- Uma subclasse também pode estender o comportamento da superclasse

```
class Automovel
{
    //...
}
```

```
class Onibus : Automovel
{
    public void AbrirPorta()
    {
        //...
    }
}
```

```
Onibus o = new Onibus();
o.Acelerar();
o.AbrirPorta();
```

Além dos elementos definidos em **Automovel**, o objeto **Onibus** também tem acesso aos seus próprios elementos

---

---

---

---

---

---

---

---

## Modificador *protected*



- Em situações onde a herança é utilizada o modificador **protected** pode ser aplicado
- Um elemento declarado com visibilidade *protected* pode ser acessado
  - Pela classe que o declarou
  - Pelas subclasses

```
class Funcionario
{
    protected double salario;
}
```

A recomendação é definir fields como **private**, enquanto properties e métodos podem ser **protected**

O field é acessível também pelas subclasses

---

---

---

---

---

---

---

---

## A Classe *System.Object*



- Todas as classes em C# herdam, num último nível, de **System.Object**
  - A palavra-chave **object** representa esta classe
- Quando a herança não é especificada, a herança de *object* é assumida de forma automática

```
class Funcionario
{
}
```



```
class Funcionario : object
{
}
```

---

---

---

---

---

---

---

---

## A Classe *System.Object*



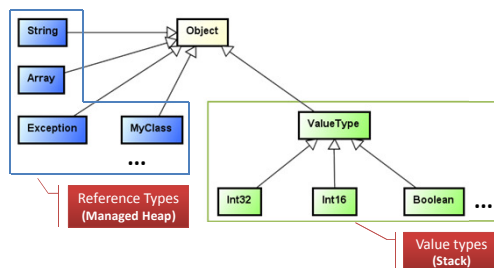
- Como todas as classes herdam de *object*, todas elas têm acesso aos métodos públicos definidos em *object*

Método	Significado
<code>string ToString()</code>	Representação do objeto em forma de texto
<code>int GetHashCode()</code>	Representado do objeto na forma de um <i>int</i>
<code>bool Equals(object o)</code>	Compara se objetos são iguais
<code>object MemberwiseClone()</code>	Faz uma cópia dos fields do objeto

## Hierarquia dos Tipos de Dados



- Os tipos de dados também são organizados numa hierarquia de classes



## Hierarquia dos Tipos de Dados



- Uma consequência interessante disso é que tipos de dados primitivos também têm acesso aos métodos de *object*

```
bool b = true;
b.Equals(true);

int x = 15;
x.ToString();

15.Equals(10);

20.32.Equals(10.10);
```

## O Modificador *sealed*



- Quando aplicado a uma classe, o modificador **sealed** não permite que ela tenha subclasses

```
sealed class Funcionario  
{  
}
```

```
class Gerente : Funcionario  
{  
}
```

Não é possível herdar de uma classe *sealed*

## Construtores e a Herança



- Cada classe tem o seu conjunto de construtores
  - Construtores não são herdados numa relação de herança
- Ao instanciar uma classe, o construtor da superclasse equivalente é sempre executado primeiro
  - O construtor de *object* é sempre o primeiro a ser executado

## Invocando Construtores



```
class Object  
{  
    public Object ()  
    { }  
}
```


```
class Automovel : Object  
{  
    public Automovel () : base ()  
    { }  
}
```

```
class Onibus : Automovel  
{  
    public Onibus () : base ()  
    { }  
}
```

```
Onibus o = new Onibus ();
```

**base()** chama um construtor da superclasse

## Invocando Construtores



```

class Object
{
    public Object ()
    { }
}

class Automovel : Object
{
    public Automovel(int p)
    { }
}

class Onibus : Automovel
{
    public Onibus() : base($0)
    { }
}

Onibus o = new Onibus();
    
```

Chama o construtor que recebe um *int* como parâmetro

---

---

---

---


---

---

---

---

## Interfaces



- São tipos de dados que contêm apenas assinaturas de métodos ou properties
  - Nada é implementado
  - Fields e construtores não são permitidos
- Tudo o que é declarado na interface, por padrão, tem visibilidade *public*

```

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

Palavra-chave interface

Métodos públicos sem implementação

---

---

---

---


---

---

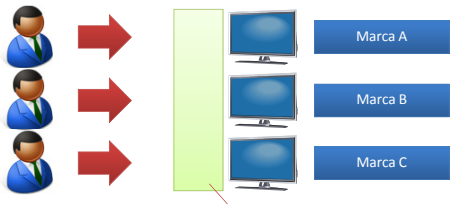
---

---

## Por que interfaces são assim?



- A interface foca no que o objeto faz, e não em como ele faz
  - O “o que” é definido pela assinatura do método
  - O “como” é definido pela implementação do método



Implementam a mesma interface

---

---

---


---

---

---

---

---

Relação entre Classes e Interfaces


- Dizemos que uma classe implementa uma interface
  - Não dizemos que ela herda de uma interface
- A classe é responsável por fornecer todas as implementações dos métodos e properties definidos na interface

```

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

    public void Desligar()
    {
    }
}
            
```

A sintaxe é semelhante à herança

A classe implementa o que a interface define

---

---

---


---

---

---

---

---

Relação entre Classes e Interfaces


- Quando uma classe implementa uma interface, ela passa a ser do tipo da interface
  - Como acontece na herança
  - Relação do tipo “é-um”

```

class Automovel : IMotorizado
{
    //...
}

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

Automovel é um Motorizado

O objeto tem acesso aos métodos da interface

Estruturas também podem implementar interfaces

---

---

---


---

---

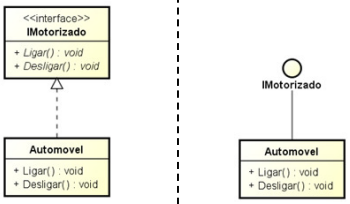
---

---

---

Interfaces em UML


- Em UML, interfaces podem ser representadas de duas maneiras



---

---

---

---

---

---

---

---



## Múltiplas Interfaces

Softblue

- Uma classe pode implementar mais de uma interface
- Basta fornecer a implementação para todos os elementos definidos em todas as interfaces

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

interface IRoubavel
{
    void Roubar();
}
```

As interfaces são separadas por vírgulas

```
class Automovel : IMotorizado, IRoubavel
{
    public void Ligar() { }
    public void Desligar() { }
    public void Roubar() { }
}
```

---

---

---

---

---

---

---

---

## Herança e Implementação

Softblue

- Uma classe pode herdar de outra e implementar uma ou mais interfaces, tudo ao mesmo tempo

```
class Carro : Automovel, IMotorizado, IRoubavel
{
}
```

Superclasse (no máximo uma)

Interfaces

- Estruturas podem implementar uma ou mais interfaces, mas não possuem suporte à herança

---

---

---

---

---

---

---

---

## Herança em Interfaces

Softblue

- Uma interface pode herdar de outra interface
  - A herança pode ser de uma interface ou mais

```
interface IExemploA
{
}

interface IExemploB
{
}

interface IExemploC : IExemploA, IExemploB
{
}
```

IExemploC passa a ter as declarações dos elementos das suas superinterfaces

---

---

---

---

---

---

---

---

## Herança X Implementação



- Superclasses ou superinterfaces
  - Uma classe pode ter apenas uma superclasse direta
  - Uma classe pode implementar várias interfaces
- Membros
  - Uma classe pode ter fields, construtores, métodos e properties
  - Uma interface pode ter apenas métodos e properties
- Hierarquia
  - Uma interface não precisa estar relacionada com a hierarquia de classes

---

---

---

---

---

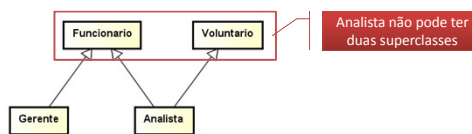
---

---

## O Problema da Hierarquia



- Exemplo
  - Todo analista, além de funcionário, é também um voluntário



---

---

---

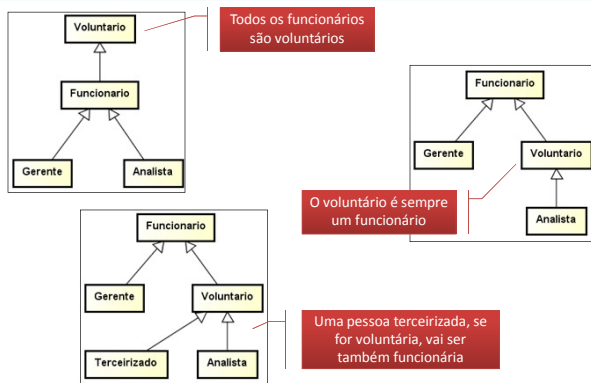
---

---

---

---

## O Problema da Hierarquia



---

---

---

---

---

---

---

## O Problema da Hierarquia

Softblue

- Transformar **Voluntario** em uma interface resolve o problema

Interfases podem ser usadas em hierarquias de classes completamente diferentes

Terceirizado é apenas voluntário

Gerente é apenas funcionário

Analista é funcionário e voluntário

---

---

---

---

---

---

---

---

## Associação

Softblue

- Além da herança, classes podem se relacionar através de associação
- Na associação, a relação é do tipo “tem-um”

```
class Salario {
    private double valor;
    private int diaPagto;
}
```

Funcionário tem um Salário

```
class Funcionario {
    private Salario salario;
}
```

A associação é estabelecida através de um field que referencia outra classe

---

---

---

---

---

---

---

---

## Associação na UML

Softblue

- Na UML, a associação é representada da seguinte forma

A seta indica a navegabilidade da associação

- Se quiser saber mais, busque informações sobre os tipos de associação e sobre multiplicidade em associações

---

---

---

---

---

---

---

---

## Exemplo de Interface: *ICloneable*



- A interface **ICloneable** pode ser implementada por classes capazes de clonar objetos

```
class Caneta : ICloneable
{
    public string Marca { get; set; }
    public bool Fechada { get; set; }
    public int Cor { get; set; }

    public object Clone()
    {
        Caneta c = new Caneta();
        c.Marca = this.Marca;
        c.Fechada = this.Fechada;
        c.Cor = this.Cor;
        return c;
    }
}
```

Implementação da interface

Método de clonagem

## Exemplo de Interface: *ICloneable*



- O método **Clone()** é chamado para clonar um objeto

```
Caneta c1 = new Caneta();
c1.Marca = "Bic";
c1.Fechada = true;
c1.Cor = 5;

Caneta c2 = (Caneta)c1.Clone();
```

Método *Clone()* gera um novo objeto. O casting é necessário.

## Exemplo de Interface: *ICloneable*



- O método **MemberwiseClone()** pode ser usado para copiar os valores de todos os fields

```
public object Clone()
{
    return this.MemberwiseClone();
}
```

Se o field for um *reference type*, *MemberwiseClone()* copiará a referência, não o objeto

## Polimorfismo



- O correto uso da herança de classes e implementação de interfaces favorece o polimorfismo
  - Polimorfismo é um dos pilares da orientação a objetos

---

---

---

---

---

---

---



**Softblue**

---

---

---

---

---

---

---