



C#: PROGRAMAÇÃO ORIENTADA A OBJETOS

Prof. Alexandre Krohn

O QUE É P.O.O.?

Objetivos da P.O.O

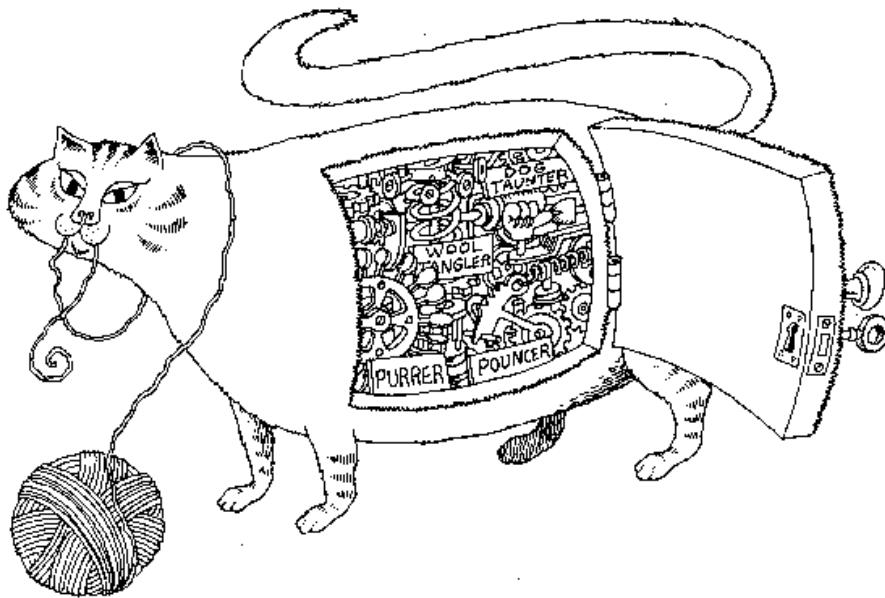
- ▢ Aproximar o mundo real do mundo virtual
 - ▢ Simular o mundo real dentro do computador
- ▢ Por isso utiliza objetos
 - ▢ O que é um Objeto?
 - ▢ Uma representação virtual de um objeto real
- ▢ Moldar o mundo dos objetos
 - ▢ Explicar as interações entre estes

Programação orientada a objetos é, portanto, uma maneira de fazer software utilizando-se abstrações dos elementos do “mundo real” presentes no domínio da aplicação.

O QUE É P.O.O?

Na programação orientada a objetos, os detalhes da implementação estão “escondidos” em um tipo abstrato de dados

- O usuário apenas visualiza as interfaces de interação.



O QUE É P.O.O.?

Vantagens

- ▢ Melhor representação do “mundo real”
- ▢ Ênfase nos dados
- ▢ Modularidade
- ▢ Reusabilidade
- ▢ Produtividade
- ▢ Manutenibilidade

Desvantagens

- ▢ Paradigma um pouco mais complicado em relação as linguagens procedurais
- ▢ Tempo de execução geralmente superior ao das linguagens procedurais

PROGRAMAÇÃO PROCEDURAL

Foco é na sequência de operações sobre um conjunto de dados para atingir um determinado resultado

Entretanto, ambos são definidos de forma separada

```
struct Circulo {  
    float raio;  
};  
  
float area(struct Circulo c) {  
    return c.raio * c.raio * 3.14159;  
}
```

A única relação entre o dado e as operações são os parâmetros de entrada ou valor de retorno

PROGRAMAÇÃO ORIENTADA A OBJETOS

O foco da POO é o objeto, o qual possui um comportamento e estados possíveis definidos através de uma Classe

- ▮ Estados possíveis são definidos pelos atributos da classe
- ▮ Comportamento é definido pelos métodos (funções) da classe

Classe é uma abstração que modela objetos do mundo real e definem os atributos e métodos (membros) de um grupo de objetos

- ▮ Atributos (estado) e Métodos (comportamento) são **encapsulados** em um tipo abstrato de dados
- ▮ Os atributos são “escondidos” dentro da classe

CLASSES

```
class Complexo
{
    // Atributos
    float real, imaginario;
    // Métodos
    public float modulo()
    {
        return (float) Math.Sqrt(real *
real + imaginario *
imaginario);
    }
}
```

CLASSES

Objetos são instâncias de classes

- Objetos podem ser criados usando a palavra-chave new seguida do nome da classe

```
Complexo c = new Complexo();
```

Acesso aos métodos é realizado através do operador “.”, ou através de operadores quando estes são definidos pela classe

```
float mod = c.modulo();
```


ATRIBUTOS

Os atributos são uma coleção de variáveis de qualquer tipo que são declarados diretamente na classe (fora de qualquer método)

```
class Complexo
{
    float real, imaginario;
}
```

A visibilidade padrão dos atributos é interna (private)

- ▮ Os atributos devem ser fornecidos através de métodos
- ▮ Este acesso indireto aos atributos permite um acesso mais controlado (validação)

MÉTODOS

Métodos definem o comportamento da classe, e são implementados através de funções

```
tipoVisibilidade tipoDeRetorno nomeDoMetodo  
( listaDeParâmetros )  
{  
    // as instruções do corpo do método ficam aqui  
}
```

onde

tipoVisibilidade define a visibilidade do método

tipoDeRetorno define o tipo a ser retornado

nomeDoMetodo define o nome do método a ser invocado

listaDeParâmetros define a lista de parâmetros

MÉTODOS

```
class Complexo
{
    public float modulo()
    {
        return (float) Math.Sqrt(real *
            real + imaginario *
            imaginario);
    }
}
```

A visibilidade de um método deve ser pública (**public**) a fim de que o objeto possa utilizá-lo

MÉTODOS: CONSTRUTOR

Sempre que um objeto é instanciado (criado), o construtor é chamado

Objetivo é inicializar os membros da classe e definir o estado inicial da instância

Método construtor possui o mesmo nome da classe

- Não possui tipo retornado (nem mesmo void)

Uma classe pode ter vários construtores que recebam argumentos diferentes.

MÉTODOS: CONSTRUTOR

Se você não fornecer um construtor para a classe, o C# criará um por padrão (sem parâmetros) que instancia o objeto e define valores padrões para as variáveis membro

Tipos de valor	Valor padrão
bool	false
Byte	0
char	'\0'
decimal	0.0M
double	0.0D
float	0.0F
int	0
long	0L
sbyte	0
short	0
uint	0
ulong	0
ushort	0

MÉTODOS: CONSTRUTOR

```
class Complexo
{
    // Atributos
    float real, imaginario;
    // Métodos
public Complexo(float r, float i)
    {
        real = r;
        imaginario = i;
    }
}
```

Quando um construtor é definido, o construtor default (sem parâmetros) não poderá ser utilizado

MÉTODOS: CONSTRUTOR

Invocando um construtor

- ▮ Ao utilizar `new` para declarar uma instância de uma classe, a memória é alocada, as variáveis são inicializadas e então o construtor é chamado
 - ▮ O garbage collector automaticamente gerencia a memória

```
Complexo c = new Complexo(2,3);
```

Caso for invocado um construtor inexistente, o compilador retornará um erro de compilação

MÉTODOS: DESTRUIDORES

Destruidores são usados para destruir instâncias de classes

- ▢ Utiliza para finalizar algum recurso utilizado pelo objeto ()

Chamado pelo garbage collector

- ▢ NUNCA chame o destruidor
- ▢ Não-determinístico (não se sabe quando vai ser chamado)

Sem modificadores de acessos e sem parâmetros

- ▢ Nome começa por ~

Uma classe só pode ter um destruidor

VISIBILIDADE DOS MEMBROS DE UMA CLASSE

Modificadores de acesso controle o encapsulamento dos membros de uma classe (atributos e métodos)

public

Qualquer um pode acessar



public



private

private (default)

Acesso limitado da classe (quando não o método não deve ser acessado fora da classe)

protected

Privado para a classe e qualquer classe derivada (herança)



protected

VISIBILIDADE DOS MEMBROS DE UMA CLASSE

Atributos

- ▮ Devem ser definidos como **private**

Métodos

- ▮ Deve disponibilizar somente os métodos que devem ser utilizados pelo usuário (**public**)
- ▮ Os demais métodos devem ser “escondidos” (**private** ou **protected**)

MÉTODOS: PROPRIEDADES

Parecem variáveis...

Utilizados como variáveis...

MAS são realmente métodos

- ▢ get & set

- ▢ Proteção

Eles podem ser utilizados para validar dados antes de permitir uma alteração

MÉTODOS: PROPRIEDADES

Pode ser utilizado como variável local ou um campo

```
class Complexo
{
    float real, imaginario;

    public float Real
    {
        get { return real; }
        set { real = value; }
    }
}
```

A palavra-chave **value** é a variável implícita passada para propriedades

□ Tipo é o mesmo do valor retornado

MÉTODOS: PROPRIEDADES

O Visual Studio possui uma opção para gerar automaticamente uma propriedade

Para gerar automaticamente a propriedade, execute os seguintes passos:

1. Coloque o cursor no atributo que deseja criar uma propriedade
2. No menu Editar, clicar na opção **Refatorar** e em seguida na opção **Encapsular Campo**. Uma janela será aberta.
 - ▮ Você pode pressionar CTRL + R e em seguida CTRL + E
3. Clicar no botão **Aplicar**

Por exemplo, no caso de uma variável chamada teste, a propriedade criada seria

```
public double Teste { get => teste; set => teste =  
value; }
```

MÉTODOS: PROPRIEDADES

Para facilitar a declaração das propriedades, a partir do C# 3.0, temos as propriedades que são implementadas automaticamente pelo compilador, as propriedades auto implementadas

```
public float Real { get; set; }
```

Toda vez que declaramos uma propriedade auto implementadas, precisamos sempre declarar um get e um set para a propriedade, porém podemos controlar a visibilidade tanto do get quanto do set.

```
// get é público e pode ser acessado por qq classe
```

```
// set é privado e por isso só pode ser usado por  
Complexo
```

```
public float Real { get; private set; }
```

RECAPITULANDO...

```
class Retangulo
```

```
{
```

```
    float altura, largura;
```

```
    public Retangulo(float a, float l)
```

```
    {
```

```
        altura = a;
```

```
        largura = l;
```

```
    }
```

```
    public float Altura
```

```
    {
```

```
        get { return altura; }
```

```
        set { altura = value; }
```

```
    }
```

```
    public float Area()
```

```
    {
```

```
        return altura * largura;
```

```
    }
```

```
}
```

Declaração da classe

Atributos

Construtor

Propriedade

Método

HERANÇA

Herança é um conceito chave usado na programação orientada a objetos para descrever uma relação entre as classes

Por meio da herança uma classe copia ou herda todas as propriedades, atributos e métodos de uma outra classe, podendo assim estender sua funcionalidade

- A classe que cede os membros para a outra classe é chamada superclasse, classe pai ou classe base
- A classe que herda os membros da outra classe é chamada subclasse ou classe derivada;

A herança permite a reutilização de código e especifica um relacionamento de especialização/generalização do tipo "é um"

HERANÇA

Uma classe herda de uma única classe base

- ▮ Especificado diretamente na declaração
- ▮ Herança circular não é permitida
- ▮ Caso não especificado, a classe herda implicitamente da classe **object**

```
class Poligono      // Classe base
{
    ...
}
class Quadrilatero : Poligono  // Classe derivada
{
    ...
}
```

Uma classe herda todos os membros não privados da classe base

HERANÇA

Uma classe herda todos os membros não privados da classe base

- ▢ Modificador de acesso **protected** torna o método ou atributo privado para a classe e qualquer classe derivada

CONSTRUTORES

Ao instanciar um objeto de uma classe derivada, o construtor da classe base também é chamado implicitamente

- O construtor default base é chamado por default

Quando a classe base não possuir construtor sem parâmetros (ou construtor default), a classe derivada deve chamar explicitamente o construtor da classe base

- Especifica qual construtor base deseja-se chamar utilizando a palavra-chave **base**

```
class Quadrilatero : Poligono
{
    public Quadrilatero() : base(4) { }
}
```

OCULTANDO MEMBROS DA CLASSE BASE

Se você quiser que o seu membro derivado tenha o mesmo nome de um membro de uma classe base, mas não quiser que ele participe da invocação virtual, você pode usar a palavra-chave **new**

▮ A palavra-chave **new** é colocada antes do tipo de retorno de um membro de classe que está sendo substituído

```
class Poligono          // Classe base
{
    public float Comprimento(int index)
    {
        return comprimentos[index];
    }
}
class Quadrilatero : Poligono    // Classe derivada
{
    public new float Comprimento(int index)
    {
        return comprimentos[0];
    }
}
```

NÃO PERMITIR HERANÇA

Quando deseja-se que uma classe não seja herdada, deve ser declarada como **sealed**

- ▢ **structs** (todos os tipos valorados) são implicitamente **sealed**

- ▢ Uma classe abstrata não pode ser **sealed**

```
sealed class ClasseSemHeranca
```

```
{
```

```
...
```

```
}
```

Uma classe **sealed** não pode ter membros virtuais

POLIMORFISMO

O polimorfismo costuma ser chamado de o terceiro pilar da programação orientada a objetos, depois do encapsulamento e a herança

POLIMORFISMO

As classes base podem definir e aplicar métodos (chamados virtuais) e as classes derivadas podem substituí-los, o que significa que elas fornecem sua própria definição e implementação.

- ▢ Em tempo de execução, quando o código do cliente chama o método, o CLR procura o tipo do objeto e invoca a substituição do método
- ▢ Dessa forma, você pode chamar em seu código-fonte um método de uma classe base e fazer com que a versão de uma classe derivada do método seja executada

POLIMORFISMO: MÉTODOS VIRTUAIS

Métodos definidos como virtuais em uma classe base permitem que a sua implementação seja modificada na herança em uma classe derivada

- ▢ Não pode alterar o modificador de acesso (public/private/protected)
- ▢ O método da classe base que pode ser alterado na classe derivada deve ser declarado **virtual**

```
// Classe base
class Poligono
{
    public virtual float Area() { ... }
}
```


POLIMORFISMO: MÉTODOS VIRTUAIS

O método da classe derivada que sobrescreve o método da classe base deve ser declarado como **override**

```
// Classe derivada
class Quadrilatero : Poligono
{
    public override float Area() { ... }
}
```

CLASSES ABSTRATAS

Classes abstratas servem como um “modelo” para outras classes que dela herdem, não podendo ser instanciada por si só

É uma classe incompleta, pois o objetivo é utilizá-la como uma classe base para outras classes

Os métodos que devem ser implementados pela classe derivada devem ser declarados como `abstract` e não devem possuir código

```
abstract class Poligono
{
    public abstract float Area();
}
```

CLASSES ABSTRATAS

Se uma classe possuir um método abstrato, então a classe deve ser abstrata

Classes abstratas não podem ser instanciadas

A classe que herda de uma classe abstrata deve implementar TODOS os métodos da classe abstrata

□ Deve-se utilizar o modificador `override` para sobrescrever o método

```
class Quadrilatero : Poligono
{
    public override float Area() {...}
}
```

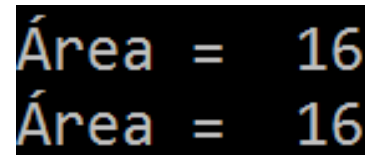
POLIMORFISMO

Em tempo de execução, os objetos de uma classe derivada podem ser tratados como objetos de uma classe base

- ▮ Quando isso ocorre, o tipo declarado do objeto não é mais idêntico ao seu tipo de tempo de execução.

```
Quadrilatero quadrado = new Quadrilatero();  
float[] comprimentos = new float[1] { 4 };  
quadrado.AtualizaComprimentos(comprimentos);  
Console.WriteLine("Área = {0} ",  
    quadrado.Area());
```

```
Poligono poligono = quadrado;  
Console.WriteLine("Área = {0} ",  
    poligono.Area());
```



Área = 16
Área = 16