cencal centro de formação profissional para a indústria de cerâmica

Programação em C#

António Ramos | Outubro 2022

Co-Financiado por:

IEFP APICER associação portuguesa das indústrias de cerâmica e cristalaria ARO Associação Empresarial da Região Oeste REPÚBLICA PORTUGUESA POCH PORTUGAL 2020 UNIÃO EUROPEIA Fundo Social Europeu

cencal centro de formação profissional para a indústria de cerâmica

Programação orientada a objectos

- Divisão do programa em vários objectos
- Interação entre classes
- Criação através de "templates"

Criação de classes

- Keyword "class"
- Convenção PascalCasing
- Conteúdo (campos, propriedades, métodos) são criados dentro da classe

```
class Pessoa {
    // Campos, Propriedades e Métodos
}
```

Campos

- Semelhante às variáveis + o "access modifier"
  - public
  - private
  - protected
  - internal

```
class Pessoa {
    private string nomeDaPessoa;
    private int numeroFiscal;
}
```

Propriedades

- Utilizadas para dar acesso a campos
- Controlo sobre o direito à informação aos campos “private”.

Declaramos as propriedades, indicando o access modifier e o data type da propriedade.

```
public int NumeroFiscal
{
    // get
    // set
}
```

Propriedades

Têm dois métodos especiais denominados como "accessors":

- Getter
  - Retorna o valor do campo
- Setter
  - Define o valor do campo

```
get {
    return numeroFiscal;
}

set {
    numeroFiscal = value;
}
```

Nota: Podemos declarar os acecessors como privados.

Propriedades auto-implementadas

Versão "short-hand" utiliza-se quando não é necessário utilizar lógica adicional ao getter e setter.

Não é necessário definir os campos “private”, pois o compilador cria-os automaticamente.

```
public int NumeroFiscal { get; set; }
```

Métodos

- Blocos de código que efetuam determinada tarefa
- Criados dentro da classe com o nível de acesso e depois o tipo de retorno

```
public void BoasVindas()
{
    Console.WriteLine("Bem-vindo!! :)");
}
```

Métodos

Podem receber parâmetros e retornar dados:

```
public double CalculaCubo(double valor)
{
    return Math.Pow(valor, 3);
}
```

Overloading

Utilização de métodos com o mesmo nome, mas com assinaturas diferentes.

```
public void BoasVindas(string nome)
{
    Console.WriteLine("Bem-vindo {0}!! :)", nome);
}
```

Construtores

- Métodos especificos para a criação de objetos do tipo da classe
- 1º método a ser chamado para a criação de objetos e usado para inicializar os métodos e as variáveis.
- Deve ter o nome da Classe e não retornar nenhum valor.

```
public Pessoa(int nif)
{
    numeroFiscal= nif;
    Console.WriteLine($"NIF = {numeroFiscal}");
}
```

Instanciar objectos

Devemos instanciar as classes no nosso programa para a utilização dos objetos

```
NomeClasse nomeObjectoX = new NomeClasse(argumentos);
```

Keyword "static"

Usada para a criação de objectos sem a criação de objectos.

Nota: Em classes estáticas todos os membros da classe devem ser estáticos

Parâmetros (Tipos de Valor VS Tipos de Referência)

Tipo de valor:

Na passagem de uma variável para o método, qualquer alteração dessa variável será feita nesse contexto.

Quando o programa sai do método, a alteração deixa de ser válida.

Parâmetros (Tipos de Valor VS Tipos de Referência)

Tipos de Referência:

Se passamos a referência do variável, qualquer alteração é válida, mesmo depois do término do método.

Cont.

```
public void PassagemPorValor(int a)
{
  a = 10;
  Console.WriteLine("valor de a = {0}", a);
}

public void PassagemPorReferência(int[] b)
{
  b[0] = 5;
  Console.WriteLine("valor de b[0] = {0}", b[0]);
}
```

Parâmetro "ref"

Para passar as variáveis do tipo de valor como referência

```
public void PassagemPorValor(ref int a)
{
  a = 10;
  Console.WriteLine("valor de a = {0}", a);
}
```

Herança

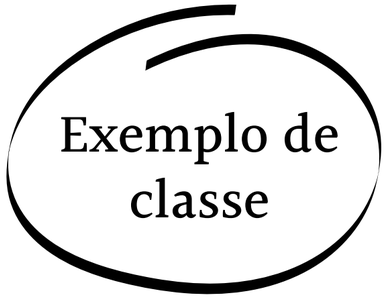
Permite a criação de uma classe, a partir de uma classe já existente.

Permite:

- Extensão de funcionalidades
- Reutilização de código
- Modificação do comportamento definido em outras classes.

Herança

A classe cujos membros são herdados é definido como classe “base” e a classe que herda esses membros é classe “derivada”.




```
class Membro
{
  protected int feeAnual;
  private string nome;
  private int membroID;
  private int membroDesde;
}
```

Classes derivadas (Subclasses)

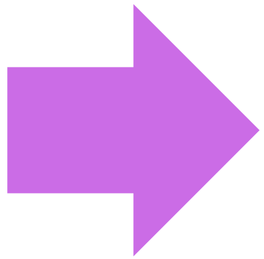
Herdam todas os membros públicos e “protected” da classe Mãe (“base”).

Para indicarmos que a classe é derivada de outra, usamos os dois pontos (:) a seguir ao nome da classe derivada.

Classes derivadas (Subclasses)

```
class Membro
{
  protected int feeAnual;
  private string nome;
  private int membroID;
  private int membroDesde;
}
```

Classe base



```
class MembroVIP : Membro
{

 }
```

Classe derivada

Polimorfismo

Conceito de dar várias formas aos métodos das classes derivadas.

Isto é, o mesmo método com a mesma assinatura pode ter várias implementações nas classes derivadas.

Polimorfismo

Para indicarmos que o nosso método da classe base é passível de ser alterado nas classes derivadas, usamos a keyword “virtual”.

Na classe derivada usamos a keyword “override” no método com a implementação diferente da classe base.

Polimorfismo

```
1  public class Veiculo{
2      public string Tipo {get;set;}
3
4      public Veiculo(string tipoVeiculo)
5      {
6          this.Tipo = tipoVeiculo;
7      }
8
9      public virtual void Mover(){
10     }
11     public virtual void Parar(){
12     }
13 }
14
```

```
20 public class Carro : Veiculo {
21
22   public override void Mover(){
23     Console.WriteLine("Carro anda!");
24   }
25
26   public override void Parar(){
27     Console.WriteLine("Carro pára!");
28   }
29 }
```

Polimorfismo

```
11 public class Veiculo{
12     public string Tipo {get;set;}
13 
14     public Veiculo(string tipoVeiculo)
15     {
16         this.Tipo = tipoVeiculo;
17     }
18 
19     public virtual void Mover(){
20     }
21     public virtual void Parar(){
22     }
23 }
```

```
25 public class Carro : Veiculo {
26   public Carro(string tipoVeiculo) : base(tipoVeiculo){
27   }
28 
29   public override void Mover(){
30     Console.WriteLine("Carro anda!");
31   }
32 }
```

Exemplo de construtor com parâmetros

Polimorfismo

Existe a possibilidade de definir uma lista das classes mãe, instanciando objetos das classes derivadas.

```
Membro[] membrosGinasio = new Membro[5];
membrosGinasio[0] = new MembroNormal("Special Rate", "Jose", 30);
membrosGinasio[1] = new MembroVIP("Carolina", 20);
membrosGinasio[2] = new MembroVIP("Maria", 21);
```

Métodos e classes "Abstratas"

São utilizadas apenas para servirem de base às classes derivadas. Estas não podem ser instanciadas.

Podem conter campos, propriedades e métodos como as outras classes, excepto que não podem ter membros estáticos.

```
abstract class AMinhaClasseAbstrata
{
}
```

Métodos e classes "Abstratas"

Os métodos abstratos deverão ser criados obrigatoriamente na declaração das classes derivadas.

Os métodos podem conter lógica e pode ser declarado um construtor na classe abstrata.

```
public abstract void MeuMetodoAbstrato();
```

Nota: Os métodos abstratos ao contrário dos outros, devem conter o ponto e virgula no final da sua declaração.

Métodos e classes "Abstratas"

Estes podem ser utilizados para o desenho de componentes, com funcionalidades pré-estabelecidas ou até utilizados para a criação de funcionalidades comuns ao longo das classes derivadas.

```
13  abstract class Animal {
14      public abstract void fazBarulho();
15  }
16
17  class Cao : Animal {
18      public override void fazBarulho() {
19        Console.WriteLine("Ão Ão");
20      }
21  }
```

```
5       public static void Main()
6       {
7             Cao obj = new Cao();
8             obj.fazBarulho();
9
10            Console.ReadLine();
11      }
```

Interfaces

Semelhante às classes abstratas, estas não são instanciadas e só podem ser "herdadas".

Apenas contém as declarações e não possuem lógica. Todos os métodos existentes nas interfaces devem se obrigatoriamente especificados nas classes derivadas.

Interfaces

As classes podem conter várias “interfaces” mas apenas podem herdar uma superclasse.

É aconselhável e é boa prática identificar as classes interface iniciando com a letra I.

Interfaces

```
public interface ILogicaNegocio
{
  void Inicializacao();
}
```

```
public class LogicaNegocio : ILogicaNegocio
{
  public void Inicializacao()
  {
      //Some code
  }
}
```

Enum

Permite ao programador utilizar nomes friendly para constantes numerárias.

Para declarar usa-se a keyword “enum” antes do nome da variável, sendo os valores passados dentro dos parênteses separados por virgula

```
enum WeekDayNumber
{
 Sun = 1, Mon = 2, Tues = 3, Wed = 4, Thurs = 5, Fri = 6, Sat = 7
}
```

Struct

Representam estruturas de dados, podem conter campos, métodos, propriedades como as classes.

Estes são utilizados para definir estruturas de dados complexas, pares de chave-valor, entre outros.

```
public struct Money
{
  public string Currency { get; set; }
  public double Amount { get; set; }
}
```

```
Money money = new Money();
Console.WriteLine(money.Currency);
Console.WriteLine(money.Amount);
```

Dictionary<TKey,TValue>

Permitem criar listas/coleções genéricas de pares valor-chave. As suas chaves devem ser únicas e os seus valores são acessíveis pela chave myDictionary[key]

Podem ser utilizados os métodos Add(), Remove(), Update() para manipular as coleções ao longo do programa.

Dictionary<TKey,TValue>

```
// Create a new dictionary of strings, with string keys.
Dictionary<string, string> openWith =
  new Dictionary<string, string>();

// Add some elements to the dictionary. There are no
// duplicate keys, but some of the values are duplicates.
openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");
openWith.Add("dib", "paint.exe");
openWith.Add("rtf", "wordpad.exe");
```