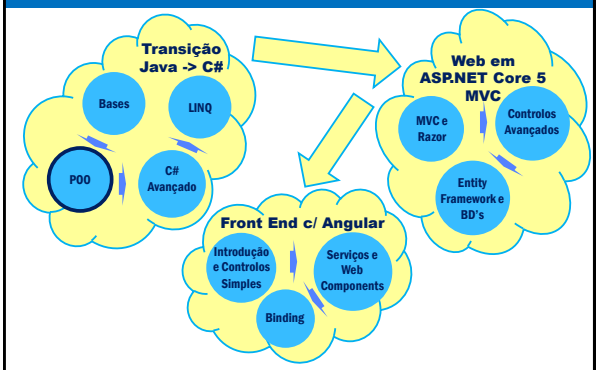
# Programação Visual

C# - Relacoes

1

## PROGRAMA






2

## C# - Relações

- **Composição (Has-A / Tem)**
  - Relação "Ter". Uma classe TEM objetos de outra(s) classes.
- **Herança (Is-A / É)**
  - Relação "Ser", Uma classe é a superclasse. Reutilização massiva de código e polimorfismo
- **Polimorfismo**
  - Uma entidade com diferentes comportamentos consoante o papel que encarna
- **Classes Abstratas**
  - Tipos de dados não instanciáveis, úteis para agrupar código e permitir polimorfismo.
- **Interfaces**
  - Tipos de dados que contêm apenas métodos abstratos (e, em c# propriedades …)
- **static**
  - Membros de Classe – São da classe: um mesmo elemento partilhado todos os objetos
- **sealed**
  - Selar classes e métodos para impedir herança
- **CONSTANTES**
  - const e readonly
- **Try / Catch**
  - Tratamento de erros e lançamento de exceções



3

## C# - relação de Composição

O conceito é simples

```
class FiguraGeometrica
{
   // RELAÇÃO DE COMPOSIÇÃO ENTRE CLASSES
   // uma classe TEM objetos de outras classes
   // uma classe É COMPOSTA por objetos de outras classes
   // A Classe FiguraGeometrica É COMPOSTA por um Point2D
   // A Classe FiguraGeometrica tem como atributo
   // um objeto da classe (do tipo de dados) Point2D

   private Point2D origem;
}
```

Vamos ver as consequências no VS



4

## C# - relação de Herança

Vamos ver as consequências no VS

```
class Quadrado : FiguraGeometrica
{
   // RELAÇÃO DE HERANÇA ENTRE CLASSES
   // uma classe É uma outra classe
   // uma classe herda
   // todos os membro da
   // classe base …(a super-classe do java)
   // A Classe Retangulo É uma FiguraGeométrica
   // A Classe Retangulo herda de FiguraGeométrica
}
```

O conceito é simples



5

## C# - relação de Herança

- Exemplos Práticos:
  - Usar a herança
    - **Operador :**
  - Podem existir membros 'protected'
    - **só acessíveis para as subclasses (da mesma ou outras assembly)**
  - Usar o construtor da classe base
    - **public SubClassConstructor() : base ( )**
  - Palavras reservadas (em c# são obrigatórias):
    - **virtual** (permite que um método seja reescrito/overrided numa subclasse)
    - **override** (substitui o método virtual da classe base (superclasse)
    - **new** (cria um novo método com o mesmo nome na subclasse)
    - **abstract** (para declarar métodos sem definição/corpo -> obriga a que a classe também seja abstract)
  - mais sobre keywords em:
    - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/



6

## C# - Polimorfismo

- **Principio da substituição**
  - Uma classe define um **tipo logo** uma subclasse define um **subtipo.**
  - Sempre que é necessário um objeto de um tipo pode usar-se em vez dele um objeto de um subtipo.
  - Ex: se Retangulo herdar de FiguraGeometrica (Retangulo é subclasse de FiguraGeometrica)

```
FiguraGeometrica fg = new Retangulo ();
```

- **Polimorfismo**
  - Um método **várias** (poli) **formas** (Morfo) de resposta
  - Quando pedimos a uma referência para executar um método, o método que é executado é o da classe do objeto referenciado.
  - Ex: Se em fg estiver um Retângulo o método Movimentar executado é o da classe Retângulo, se lá estiver uma FiguraGeometrica o Movimentar executado é o da classe FiguraGeometrica.

```
fg.Movimentar(2,3);
```



7

## C# - Polimorfismo

```
public class Aluno : Pessoa {
   private string ipsId;

   public string getId()
   {
      return ipsId;
   }
}

public class Pessoa {
   private string numeroCC;

   public string getId()
   {
      return numeroCC;
   }
}
```

```
Pessoa[] pessoas;
pessoas = new Pessoa[4];

Aluno a1 = new Aluno("2022216023");
Aluno a2 = new Aluno("2022216024");
Pessoa p1 = new Pessoa("675432123");
Pessoa p2 = new Pessoa("987567342");

pessoas[0] = a1;
pessoas[1] = p1;
pessoas[2] = a2;
pessoas[3] = p2;

foreach(Pessoa p in pessoas)
{
   p.getId();
   Console.WriteLine(p);
}
```



8

## C# - Classes abstractas

- Classes Abstratas
  - São classes que não podem ser instanciadas.
  - São úteis para agregar código e **permitir polimorfismo**
- Tornar uma classe abstracta
  - Usar a keyword 'abstract'
  - Impede a criação de objetos dessa classe
  - Podem ter métodos abstratos (não possuem implementação)
    - Usam o modificador abstract
    - Possuem apenas a declaração
    - São automaticamente virtuais



9

## C# - Interfaces

- **O que são?**
  - São tipos de dados que não podem ser instanciados e que contêm apenas declarações de métodos.
  - No design são úteis para definirem a interface de uma classe.
  - Na implementação são úteis para permitirem polimorfismo mesmo quando não são possíveis relações de herança entre diferentes tipos de dados (classes)
  - Os seus métodos são públicos e abstratos por definição
  - Classes que declarem implementar uma interface implementam todos os métodos existentes nessa interface.
  - As classes podem implementar mais do que uma interface.
  - As interfaces podem herdar (apenas) de outras interfaces.
- Definir uma **Interface**

```
Interface IMovimentavel {void Movimentar(int dx, int dy);}
```

- Usar uma **Interface**
  - Implementar o método `void Movimentar (int dx, int dy){ // código }`



10

## C# - Interfaces

- **Definição**

```
Interface IMovimentavel {
   void Movimentar(int dx);
}
```

- **Implementação**

```
public class Aluno : IMovimentavel{
   private int x,y;
   public void Movimentar(int dx) {
      x +=dx;
   }
}
public class Mesa : IMovimentavel{
   private int x,y;
   public void Movimentar(int dx) {
      x += dx;
      y += dx;
   }
}
```

- **Utilização**

```
IMovimentavel[] movimentaveis;
movimentaveis = new IMovimentavel[4];

Aluno a1 = new Aluno();
Aluno a2 = new Aluno();
Mesa m1 = new Mesa();
Mesa m2 = new Mesa();

movimentaveis[0] = a1;
movimentaveis[1] = a2;
movimentaveis[2] = m1;
movimentaveis[3] = m2;

foreach(IMovimentavel m in movimentaveis)
{
   m.Movimentar(4);
   Console.WriteLine(m);
}
```



11

## C# - Selar classes e métodos

- Selar uma classe
  - Usar a keyword **sealed**
  - Impede a definição de sub-classes
- Selar um método
  - Usar a keyword **sealed**
  - Impede o override desse método
    - Apenas para métodos virtuais



12

## C# - Membros de Classe - static

```
class Ponto :
{// Atributo e método de Classe
   // Só existe um para todos os objetos da classe
   // Antecedido pela keyword static
   private static int pontosCriados;
   public static int GetPontosCriados(){return pontosCriados};
   public Ponto(int x, int y){ this.x = x;
                               this.y = y;
                               pontosCriados++;}
   //[…] omitido restante código da classe
}
static void Main(string[] args){
   Ponto p1 = new Ponto(1, 2);
   // invocamos um método de classe (static) fora da classe
   // com NomeDaClasse.NomeDoMetodo()
  Console.Write( Ponto.GetPontosCriados() + " - ");
  Ponto p2 = new Ponto(3, 4);
  Console.Write( Ponto.GetPontosCriados());
}
```

Experimente apagar a keyword static e veja se consegue contar os pontos criados

Resultado 1 - 2
Com static Conta os Ponto(s) criados !

Resultado 1 - 1
Sem static não conta os Ponto(s) criados !



13

## C# - Constantes – const

```
class Ponto
{
  // Constantes - const
  // São de classe por definição
  // não são alteráveis no programa
  public const double PI = 3.1415926;
}

static void Main(string[] args)
{
   Console.WriteLine("XXXX SLIDE 5 Constantes - keyword const");
   Console.WriteLine("Pi=" + Ponto.PI);
   // Console.WriteLine("Pi=" + p1.PI);
   // Console.WriteLine("Pi=" + Ponto.PI++);
```

ERRO - é de classe

ERRO - não pode ser alterada



14

## C# - Não alteráveis - readonly

```
class Ponto {
   // Variáveis não alteráveis - readOnly
   // são de objeto/instancia e só são alteraveis no construtor
   // logo podem ser recebidas como parâmetro no construtor
   public readonly double OUTROPI = 3.1415;
   public Ponto(int x, int y, double outroPi){
      X = x; Y = y;
      OUTROPI = outroPi;
   }
}

static void Main(string[] args){
   Console.WriteLine("XXXX SLIDE 6 Não alteráveis - readonly");
   Ponto p3 = new Ponto(1, 2, 3.1416);
   Console.WriteLine("OutroPi=" + p3.OUTROPI);
   Ponto p4 = new Ponto(11, 22, 3.1415926);
   Console.WriteLine("OutroPi=" + p4.OUTROPI);
   // Console.WriteLine("Pi=" + p4.OUTROPI++);
```

Foi inicializada com 3.1415

É alterada no constructor para o valor recebido como parâmetro

ERRO - não pode ser alterada



15

## C# - Tratamento de erros e excepções

- Solução O.O. para o tratamento dos erros:
  - Separar o código do programa do código de tratamento dos erros - Blocos try-catch (try code catch exception)

```
try {
   ...
   int res = fazerCalculo(x,y);
{
   catch( System.Exception caught )
{
   ...
}

int fazerCalculo( double x, doubley ){
   …
   if( erro )
      throw new System.ArithmeticException("Erro em fazerCalculo")
   …
      return x+y;
}
```



16