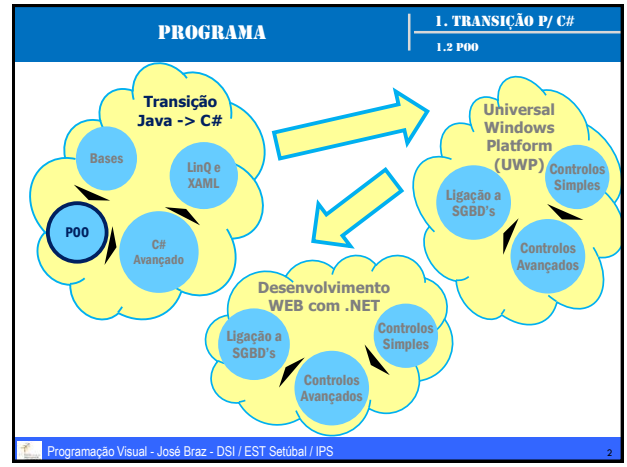
Programação Visual

C# - POO

1



2

C# - P.O.O.

- Classes
  - Tipos de Dados. Atributos sempre ***private***, construtores em que só o principal tem código
- Tipos por valor e por Referência
  - Por valor armazenam o valor, por Referência armazenam o endereço do objeto
- Métodos - Passagem de parâmetros
  - Por omissão é por valor, se antecedido de ref ou out é por referência
- Boxing/Unboxing
  - Encaixotar -> converter em referência, Desencaixotar -> converter em tipo por valor
- 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
- Interfaces
  - Tipos de dados que contêm apenas métodos abstratos, úteis para permitir o polimorfismo mesmo onde a herança não é possível.
- Polimorfismo
  - O segundo pilar da religião pê-ó-ó-ista. Permite" mandar" diferentes entidades fazerem "uma coisa" sem termos de saber quem são as entidades ou mesmo como é que o vão fazer.
- Classes Abstratas
  - Tipos de dados que não podem gerar objetos, úteis para agrupar código e permitir o polimorfismo

3

C# - Classes

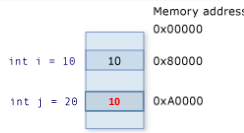
- **Campos / *Fields* (atributos)**
  - BPP: São sempre privados
  - BPP: são sempre inicializados no construtor
  - (O C# também permite inicializa-los na declaração mas seguimos as BPP)
- **Métodos**
  - Código definido dentro da classe
  - Podem ter várias formas com o mesmo identificador (polimorfismo de métodos ou sobrecarga de métodos ou overload)
- **Construtores**
  - O construtor por omissão **deve sempre** inicializar todos os 'campos'
  - Podem existir outros construtores, podendo ser então necessário definir o construtor sem argumentos
  - **Só escrevem o código para o construtor com mais parâmetros**, todos os restantes reutilizam esse construtor com valores por defeito. (ver construtores da classe Ponto)

4

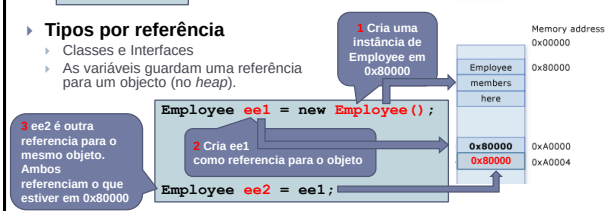
C# - Tipos valor / referência

- **Tipos por valor:**
  - int, char, double, bool, string, UInt32, Char, Double, Bool, String
  - Copy-by-value

```
int i = 10;
int j = 20;
j = i;
```



- **Tipos por referência**
  - Classes e Interfaces
  - As variáveis guardam uma referência para um objecto (no *heap*).



5

C# - Métodos – passagem de parâmetros

- **Passagem por valor –**
  - É feita por **cópia** dos valores passados.
  - É a forma de passagem de parâmetros por omissão em C#

```
void NaoTroca(int arg1, int arg2){
  int temp = arg1;
  arg1 = arg2;
  arg2 = temp;
}
```

```
int x1, x2;
x1 = 1;
x2 = 2;
Console.WriteLine("x1={0} x2={1} ", x1, x2);
        1     2
NaoTroca(x1, x2);
Console.WriteLine("\nx1={0} x2={1} ", x1, x2);
```

```
x1=1 x2=2
x1=1 x2=2
```


6

C# - Métodos – passagem de parâmetros

- **Passagem por valor**
- No caso de se passar:
  - **Um valor**: o argumento recebe uma cópia do valor.
    - Alterações no argumento não afectam o valor da variavel passada.
  - **Uma referência**: o argumento **é uma cópia da referência** (do endereço de memória onde está a variável)
    - **Alterações ao (estado do)** objecto referenciado pelo argumento **afectam** o (estado do) objecto que está na referência passada,
    - Mas **alterações do argumento não afetam** a referência que foi passada.
      - Ou seja pode pôr-se o argumento a referenciar outro objecto que a referencia que estava na variável passada não é alterada não ficando a referenciar o novo objeto.

7

C# - Métodos – passagem de parâmetros

- Passagem por **referência**
  - Possível em C# utilizando a keyword **ref** ou a keyword **out** antes do tipo do argumento
  -
  - Neste caso o argumento é o endereço de memória da variavel passada e consequentemente:
    - Quaisquer alterações no argumento feitas pelo método repercutem-se na variável passada.
  - **out** – permite passar um argumento não inicializado que terá de o ser obrigatoriamente no interior do método.
  - Nota: **ref** e **out** têm de ser usados igualmente na chamada dos métodos antes das variáveis que afectam

8

C# - Métodos – passagem de parâmetros

```
void Troca(ref int arg1, ref int arg2){
   int temp = arg1;
   arg1 = arg2;
   arg2 = temp;
}
```

```
void NaoTroca(int arg1, int arg2){
   int temp = arg1;
   arg1 = arg2;
   arg2 = temp;
}
```

```
int x1, x2;
x1 = 1;
x2 = 2;
Console.WriteLine("x1={0} x2={1} ", x1, x2);
        1      2
NaoTroca(x1, x2);
Console.WriteLine("\nx1={0} x2={1} ", x1, x2);
        x1     x2
Troca(ref x1, ref x2);
Console.WriteLine("\nx1={0} x2={1} ", x1, x2)
```

```
x1=1 x2=2

x1=1 x2=2

x1=2 x2=1
```


9

C# - Boxing e Unboxing

- **Boxing / encaixotar**
  - 'Transforma' uma variável simples (por valor) numa .Net equivalente *(por referencia)*

```
EX: int v = 23;
    UInt32 r = (UInt32) v;  // boxing explicito
    Object o =  v;          // boxing implicito
```

- **Unboxing / desencaixotar**
  - Operação inversa.
    - *EX:* `int k = (int)o;  // unboxing explicito`
- Mais em:
  - https://stackoverflow.com/questions/2111857/why-do-we-need-boxing-and-unboxing-in-c

10

C# - Boxing e Unboxing

- **Subtilezas:**

```
double e = 2.718281828459045;
double d = e;
object o1 = e;
object o2 = d;
Console.WriteLine(d == e);   //true
Console.WriteLine(o1 == o2); //false

double e = 2.718281828459045;
object o1 = e;
object o2 = e;
Console.WriteLine(o1 == o2);      // False
Console.WriteLine(o1.Equals(o2)); //True
```

```
[struct Point {
   public int x, y;
   public Point(int x, int y)
   {
      this.x = x;
      this.y = y; }
   }

Point p = new Point(1, 1);
object o = p;
p.x = 2;
Console.WriteLine(((Point)o).x);
```


11

C# - relação de Composição

```
class FiguraGeometrica
{
  // RELAÇÃO DE COMPOSIÇÃO ENTRE CLASSES
  // uma classe TEM um objeto de outra classe
  // uma classe É COMPOSTA por objetos de outras classes
  // A Classe FiguraGeometrica É COMPOSTA por um Point2D

  private Point2D origem;

  // Atributos do Java chamam-se campos (fields) em c# …
  // sempre privados camelCase
}
```


12

C# - relação de Herança

```
class Quadrado : FiguraGeometrica
{
  // RELAÇÃO DE HERANÇA ENTRE CLASSES
  // uma classe É uma outra classe
  // uma classe herda
  // todos os membro da super-classe
  // A Classe Quadrado É uma FiguraGeométrica
  // A Classe Quadrado herda de FiguraGeométrica
}
```


13

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):
    - **abstract** (para declarar métodos sem definição/corpo)
    - **virtual** (permite que um método seja reescrito/overrided numa subclasse)
    - **override** (substitui o método virtual da superclasse)
    - **new** (cria um novo método com o mesmo nome na subclasse)
  - mais sobre keywords em:
    - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/

14

C# - Interfaces

- **Interfaces**
  - 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);}

15

C# - Interfaces

Definição

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

Implementação

```
public class Aluno : IMovimentavel{
  private string nome;
  private int x,y;

  public void Movimentar(int dx) {
    x +=dx;
  }
}
```

Utilização

```
IMovimentavel[] movimentaveis;
movimentaveis = new IMovimentavel[3];

Aluno a1 = new Aluno("a1", 1);
Aluno a2 = new Aluno("a2", 2);
Aluno a3 = new Aluno("a3", 3);

movimentaveis[0] = a1;
movimentaveis[1] = a2;
movimentaveis[2] = a3;

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


16

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);`

17

C# - Polimorfismo

```
public class Aluno : IMovimentavel{
  private string nome;
  private int x;

  public void Movimentar(int dx) {
    x +=dx;
  }
}

public class Mesa : IMovimentavel{
  private int sala;
  public void Movimentar(int sala)
  {
    this.sala = sala;
  }
}
```

```
IMovimentavel[] movimentaveis;
movimentaveis = new IMovimentavel[3];

Aluno a1 = new Aluno("a1", 1);
Aluno a2 = new Aluno("a2", 2);
Mesa m1 = new Mesa(a1);
Mesa m2 = new Mesa(a2);

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

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


18

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

19

C# - Selar classes e métodos

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

20

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;
}
```


21