Programação Visual

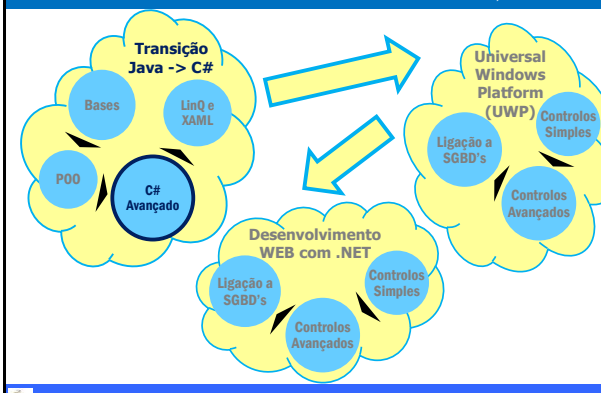
C# - Avançado 1

1

PROGRAMA

1. TRANSIÇÃO P/ C#

1.3 C# AVANÇADO



2

C# - Tópicos

- **Membros de Classe (static) versus os nossos conhecidos de objeto**
  - Métodos e Atributos de Classe (keyword **static**)
- **Constantes**
  - Keyword const e readonly
- **Propriedades**
  - Públicas, substituem os get/set,
- **Inicialização de propriedades na criação do objeto**
  - Ponto p = new Ponto() { X=1; Y=2 } ;
- Indexadores (Indexers)
  - Para acesso a arrays dentro da classe pelo identificador do objeto
- **Delegates**
  - Tipo de métodos e invocação segura de delegates.
- **Funções Anónimas**
  - Instrução inline usada sempre que seja esperado um delegate
    - Métodos Anónimos (métodos sem nome, usados até ao C# 3.0)
    - **Expressões Lambda (métodos sem nome, usados a partir do C# 3.5)**
- Syntatic sugar
  - Var – variáveis de tipo implícito
  - Tipos nullable (?)
  - Operador null-coalescing (??)
- Métodos de extensão
  - Para acrescentar métodos a classes seladas.
- Redefinição de Operadores
  - Para usar operadores quando não estão definidos pelo c#
- Evolução da sintaxe dos delegates
- **Events**

História do C#: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history

3

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 !

4

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

5

C# - Não alteráveis - readonly

```
class Ponto {
   // Variáveis não alteráveis - readOnly
   // são de objeto/instancia e só são alteraveis
   // 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;
   }
}
```

Foi inicializada com 3.1415

É alterada no constructor para o valor recebido como parâmetro

```
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=" + Ponto.Pi++);
```

ERRO - não pode ser alterada

6

C# - Propriedades

- Permitem implementar o 'acesso uniforme' a atributos referido na POO mantendo o **encapsulamento** (primeiro pilar da POO ;-)

```
//Definição
class Pessoa {
   public int Idade { get; set; }
   public String Nome { get; set;}

   public Pessoa (int idade, String nome{
      Idade = idade;
      Nome = nome;
    }
}
//Utilização
class Program {
  static void Main(string[] args){

      Pessoa p = new Pessoa(0, "S/N");
      p.Idade = 23;
      Console.WriteLine (p.Idade);
  }
}
```


7

C# - Propriedades

- Uniformizam o acesso aos atributos privados passando a tratá-los como se de variáveis de tratassem
- É uma forma de aceder a campos privados através de pseudo-métodos get e set usados 'implicitamente'
- Funcionam como se fossem um campo público mas preservam o encapsulamento

```
// definição
// Propriedade com atributo explicito
private String nome; // atributo explicito
public String Nome {
       get{return nome;}
       set{nome = value;}
}
// definição
// Propriedade com atributo implicito
public int Idade { // atributo implicito
       get;
       set;
}
//Utilização
Pessoa p1 = new Pessoa();
p1.Nome = "O nome de alguém";
String oSeuNome = p1.Nome;
p1.Idade = 20;
int x = p1.Idade;
```


8

C# - Propriedades

- Podem ser apenas de leitura,

```
public int MyReadProperty {
   get;
}
```

- Ou apenas de escrita

```
public int MyReadProperty {
   get;
   private set;
}

private int mySetOnlyField;
public int MySetProperty {
   set {mySetOnlyField = value;}
}
```


9

C# - Propriedades

- **Restrições:**
  - Não permitem a inicialização indirecta de campos
  - Não podem ser passadas por referência (ref ou out)
  - Não podem ser declaradas como 'readonly' ou 'const'
  - Não podem referir simultaneamente dois ou mais parâmetros
- **Possibilidades:**
  - Podem ser **static (de classe – só existe uma e é "partilhada" por todos os objetos da classe)**
  - Podem ser usadas em Estruturas
  - Podem ser usadas em Interfaces
  - Podem ser declaradas abstract
  - Podem ser declaradas virtual nas classes

10

C# - Inicialização de Propriedades c#

- Sintaxe própria do c# que permite inicializar atributos num constructor que não os receba como parâmetros.

Sintaxe:

Depois da lista de parâmetros **()** abrem-se chavetas **{** inicializam-se as propriedades e fecham-se chavetas **}** o ponto e virgula coloca-se depois do fecho da chaveta **;**

```
//DEFINIÇÃO
public class Pessoa
{
   public String Nome { get; set; }
   public String Telefone { get; set; }
   public bool Contactavel { get; set;}
   // Note-se a ausência
   // de construtor definido
 }

//UTILIZAÇÃO
Pessoa p = new Pessoa(){
      Nome = "Nome da pessoa",
      Telefone ="000000001"};
```


11

C# - Indexers

- Permitem usar a notação dos *arrays* para aceder a uma coleção de uma classe
- Definem-se de forma semelhante às propriedades
  - Um indexer pode referir mais do que um parâmetro como é feito em *arrays* multidimensionais.
  - Os parâmetros usados nos índices dos *indexers* podem ser de qualquer tipo que implemente a interface Enumerable.
  - No exemplo mostra-se uma classe **Ecra** com um *indexer* contendo 2 índices inteiros.

```
//DEFINIÇÃO
public class Ecra
{
  private char[,] ecra =
              new char[25, 80];

  public char this[int x, int y]
  {
    get { return ecra[x,y]; }
    set { ecra[x,y] = value;}
  }
}

//UTILIZAÇÃO
Ecra e = new Ecra();
e[2,3] = 'c';
Console.Write(e[2,3]);
```


12

C# - Indexers

- **Restrições:**
  - Não podem ser passados por referência (ref ou out)
  - Não podem ser static
  - Necessitam de pelo menos um parâmetro
- **Possibilidades:**
  - Podem ser de leitura e escrita, apenas de escrita ou apenas de leitura
  - Podem usar como indice outros tipos além do 'int'
  - Podem ter mais de um parâmetro
  - Podem ser redefinidos
  - Podem ser declarados em Interfaces
  - Podem ser 'virtual'

13

C# - delegates

- **Tipo de métodos**
  - Todos os métodos com o mesmo tipo de retorno e a mesma lista de parâmetros
- **Exemplo:**

```
void mover (int dx, int dy) {
    x+=dx; y+=dy;
}

void ampliar (int fatorX, int fatorY) {
    x*= fatorX; y *= fatorY;
}

void reduzir (int fatorX, int fatorY) {
     x = x/ fatorX; y = y/ fatorY;
}
```

São do mesmo tipo porque têm o mesmo tipo de retorno e listas de parâmetros iguais

- **delegate**
  - **É um tipo que representa referências para métodos** com uma lista de parâmetros e um tipo de retorno especificos (referencias para um tipo de métodos)
  - Permite passar referências para métodos como parâmetros de métodos.

14

C# - delegates

- **Até ao C# 3.0 A utilização de delegates passava por 3 etapas**:

1. Definição: declaração de um tipo delegate (no namespace ou na classe que o fosse usar)

Ex: `public delegate int AceitaDoubleRetornaInt(double umDouble);`

2. Criação: na classe que usa-se o delegate

- **Ex:**

```
public int MetodoPoDelegate(double d){
      return (int)Math.Round(d);
}
AceitaDoubleRetornaInt oDelegate =
      new AceitaDoubleRetornaInt(MetodoPoDelegate);
```

Com a assinatura dos métodos que lá podiamos guardar

Criávamos o delegate com um método dos que lá podiamos guardar

3. Invocação: execução de todos os métodos do delegate

Ex: `oDelegate(123.45);`

Invocamos com o nome do objeto e os valores

- Um **delegate** pode guardar mais do que um método.
- Podemos **adicionar** métodos usando o operador **+=** e **remover** usando o operador **-=**.
- Os métodos guardados nos *delegates* **podem retornar um valor** mas, neste caso, **só é devolvido o valor retornado pelo último método** que foi guardado no *delegate*.
- No caso de um dos métodos gerar uma exceção, esta propaga-se aos outros métodos já executados não sendo executado nenhum dos métodos seguintes.

15

C# - Uso de delegates no .NET 5

- **Func e Action**
  - São tipos delegate genéricos.
  - Existem 16x2 diferentes tipos de delegates (de 0 a 15 parâmetros)
  - Declaramos o nosso delegate como sendo do tipo **Func** ou **Action** parametrizado para os nossos tipos e
  - Criamo-los como lambda expressions
  - Invocamos os delegate (add e writeAdd) com parâmetros reais.

```
// vamos usar uma Func com 3 parametros
// public delegate Tresult Func
//     <T1, T2, TResult)> // tipos parametrizáveis
//     (T1 arg1, T2 arg2);// argumentos do delegate
//     Tresult é o tipo retornado por uma Func


Func <int, int , int> add =           //1 declaração
    (firstNumber, secondNumber) =>    //2 criação
     {return firstNumber + secondNumber;};

Action <int, int> writeAdd =          //1 declaração
    (firstNumber, secondNumber) =>    //2 criação
     Console.WriteLine (firstNumber + secondNumber);


int result = add(1,2);                //3
writeAdd(2,4);
Uma interessante descrição em:
http://www.silverlighthack.com/post/2008/07/14/Evolution-of-C-delegate-syntax-from-NET-1x-to-35.aspx
```


16

C# - Funções Anónimas

- Uma Função Anónima é uma instrução ou expressão "inline" que pode ser usada sempre que seja esperado um tipo delegate.
  - Pode ser usada para inicializar um determinado delegate
  - Ou ser passada como parâmetro de um método em lugar de um determinado delegate.
- Há dois tipos de funções anónimas
  - Métodos anónimos e o operador delegate
    - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/delegate-operator
  - Expressões Lambda
    - https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions

17

C# - Métodos Anónimos

- Método anónimo é uma instrução sem nome associado.
- Para definir métodos anónimos:
  - 1. Usa-se o operador **delegate**
  - 2. Seguido da lista de parâmetros
  - e do código {definição do método}.
  - Quando usamos o operador delegate podemos omitir a lista de parâmetros, caso em que o método anónimo assim criado pode ser convertido para um tipo delegado com qualquer lista de parâmetros
  - Esta é a única funcionalidade não suportada por expressões lambda

```
// Método anónimo definido
// como um delegate usando os
// templates do C#9 (.NET 5)

Action<decimal> margemDeLucro10 =
   delegate
   (decimal preco)
   {
      decimal lucro = 0.10M * preco;
      Console.WriteLine(
            "price with Profit: " +
            (lucro + preco));
   };

// executa o método anónimo
margemDeLucro10(100);
```


18

C# - Expressões Lambda

- Em C# uma expressão lambda é uma função anónima que pode ser usada para criar delegados.
- Para definir expressões lambda:
  - 1. Especifica-se a lista de parâmetros
  - 2. Seguida do simbolo =>
  - 3. Seguido do código a executar

```
// Até ao C# 7 teriamos de
// definir um delegate
// delegate void priceWithProfit (decimal price);
// priceWithProfit percentMargin50LE =


// No C#9(.NET 5) usamos os templates
Action<decimal> percentMargin10LE =
    (price)
    =>
    {
      decimal profit = 0.1M * price;
      Console.WriteLine
         ("price with Profit: " +
         (profit + price));
    };


// executa a expressão Lambda
percentMargin10LE (100);
```


19

C# - Variáveis de tipo implícito - var

- Em C# é possível utilizar a palavra reservada **var** em vez do tipo explicito.
- O compilador de C# encarrega-se de definir o tipo correto no momento da atribuição tendo em conta o tipo do valor atribuído.
- **Variáveis de tipo var:**
  - **Só podem ser usadas para variáveis locais**, ou seja não se podem usar em campos nem em parâmetros de métodos, nem como tipo de retorno de métodos.
  - **Têm de ser obrigatoriamente inicializadas** aquando da sua declaração e **não** podem ser inicializadas com **null**.
  - As boas práticas de programação desaconselham o seu uso em variáveis normais porque dificulta a interpretação do código.
  - **Na UC de PV só se usa var quando necessário**

```
static void Main(string[] args){
    var valor = 25;
    valor = 30;
    valor = val + 123;
    // valor = 123.45 // ERRO!
    var caracter = 'c';
    caracter = 'd';
    // caracter =" palavra" // ERRO!
    var nomes = new List<string>();
}
```


20

C# - Tipos nullable

- Variáveis dos tipos por valor (*value* types) em C# têm sempre que ser inicializadas com um valor diferente de **null**.
- Por vezes é útil que não tenham qualquer valor, indicando que o seu valor está indefinido.
- Isto é possível recorrendo aos tipos *nullable* do C#.
- Para representar um tipo *nullable* acrescenta-se um ponto de interrogação a seguir ao tipo simples. O tipos *nullable*, são na prática estruturas genéricas **Nullable<T>**, onde estão definida as propriedades **HasValue,** que indica se a variável tem um valor ou não, e **Value** que permite ir buscar o seu valor se este não for **null**.

```
// TRADICIONAL EM C#
static void Main(string[] args) {
   Nullable<int> i = null;
   if (i.HasValue)
      Console.WriteLine(i.Value);
   //ou (i)
   else
      Console.WriteLine("Null");
}
```

```
// SIPLIFICADA EM C#
static void Main(string[] args) {
   int? i = null;
   if (i.HasValue)
      Console.WriteLine(i.Value);
      //ou (i)
   else
      Console.WriteLine("Null");
}
```


21

C# - Operador null-coalescing ??

- O operador **??** (*null-coalescing operator*) verifica o operando da sua esquerda, se este for diferente de **null** retorna o seu valor, caso contrário retorna o valor do operando que está à sua direita.

```
// TRADICIONAL EM C#
int? a = null;
int b;
if( a == null )
   b = -1;
else
   b = a.Value;
Console.WriteLine(b);// output: -1
```

```
// SIPLIFICADA EM C#
int? a = null;
int b = a ?? -1;


Console.WriteLine(b);// output: -1
```


22

C# - Redefinição de operadores

Declaram-se como métodos públicos de classe (public static) onde o identificador é a palavra-chave **operator** seguida do símbolo do operador.

os argumentos do operador (parâmetros do método) são os operandos usados com esses operadores

```
//DEFINIÇÃO
public static Ponto operator + (Ponto p1, Ponto p2)
{
   return new Ponto (p1.X + p2.X, p1.Y + p2.Y);
}
// UTILIZAÇÃO
Console.WriteLine("p1: " + p1);                    // p1: (10,2)
Console.WriteLine("p2: " + p2);                     // p2: (2,3)
          p3 = p1 + p2;
Console.WriteLine("p3 = p1 + p2");
Console.WriteLine("p3 = " + p3);          // p3 = (12,5)
Console.WriteLine(" p1+p2 = " + p1+p2);  // p1+p2=(10,2)(2,3)
Console.WriteLine("(p1+p2)= " +(p1+p2)); //(p1+p2)= (12,5)
```


23

C# - Métodos de Extensão

- Necessidade:
  - Normalmente quando se pretende acrescentar um método a uma classe altera-se essa classe ou deriva-se uma nova subclasse que acrescenta esse método.
  - Alterar a classe pode não ser possível se esta fizer parte de uma biblioteca de que não temos acesso ao código fonte.
  - Da mesma forma, derivar uma classe pode não ser possível se a classe estiver selada (ou se se estiver a usar uma estrutura).
- Os **métodos de extensão** permitem acrescentar métodos a uma classe sem que seja necessário usar qualquer das opções referidas.
- Como?

24

C# - Métodos de Extensão

- Para definir um método de extensão
  - 1. Criar uma classe **estática** onde será definido o método de extensão (ou usar uma existente).
  - 2. Definir um método **estático**.
  - 3. Colocar como primeiro parâmetro do método o nome da classe à qual se está a acrescentar o método antecedido pela palavra reservada **this**.
    - A palavra reservada this só pode ser aplicada ao primeiro parâmetro.

```
//DEFINICAO
public static class MyExtensions
{
   public static bool IsNumeric(
            this String s){

      float output;
      return float.TryParse(
                  s,out output);
   }
}

//UTILIZACAO
string test = "4.0";
if (test.IsNumeric())
    Console.WriteLine("Yes");
else
    Console.WriteLine("No");
```


25

C# - Utilização de delegates

```
namespace TP03_CSharp
{
      public delegate void OnChanged(decimal umDecimal);

      class ExemploDelegate
      {
            public decimal Valor { get; set; }

            public void AlteraValor (decimal novoValor){
               Valor = novoValor;
            }
            public void ImprimeValor (decimal valor){
               Console.WriteLine(valor);
            }
      }
}
```

1. DEFINIÇÃO: Declarar o delegate no namespace da classe

Métodos do tipo OnChanged

São do tipo OnChanged porque
a. não retornam nada: void
b. aceitam um decimal como argumento

26

C# - Utilização de delegates

```
namespace TP03_CSharp
{
   class Program {
      static void Main(string[] args){
      // […] restante Código omitido
         ExemploDelegate umExemplo =
                        new ExemploDelegate() { Valor = 10 };
         OnChanged oc = new OnChanged(umExemplo.AlteraValor);
         oc(20); // umExemplo.Valor = 20
         oc += umExemplo.imprimeValor;
         oc(30);
      }
   }
}
```

2. INSRIÇÃO do objeto delegate

3. INVOCAÇÃO dos métodos que estiverem no delegate oc com o valor 20

Também podemos adicionar mais métodos ao delegate:

E então a invocação do delegate executa todos os métodos nele guardados

27

C# - events

- Para **impedir a anulação ou instanciação** directa de delegates o c# propõe o uso de events.
- A inscrição dos métodos no evento faz-se tal como para os **delegates** através da utilização dos operadores **+=** e **-=**.
- Tal como nas propriedades em que podemos definir o código a executar quando se lhe atribui um valor (bloco **set { }**) ou se lê o seu valor (bloco **get { }**), nos eventos é possível definir o código a executar quando se adiciona um método com o operador **+=** (bloco **add { }**) ou quando se remove um método com o operador **-=** (bloco **remove { }**).
- Uma discussão das diferenças entre delegate e event pode ser encontrada em:
  https://www.tutorialsteacher.com/articles/difference-between-delegate-and-event-csharp
- Sobre a diferença entre events com add/remove implicito e explicito:
  https://stackoverflow.com/questions/1015166/c-event-with-explicity-add-remove-typical-event/1015189
- Sobre a utilização de events com add/remove explicito:
  https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-implement-interface-events

28

C# - Utilização de events

```
namespace TP03_CSharp
{
  class ExemploEvent
  {
    public event Action<int, int> MyEvent;

    public void ExemploDeUsoDeEvent (int i, int j){

      MyEvent += MetodoParaOMyEvent;
      MyEvent += delegate (x, y) {
                 Console.WriteLine("Do Método Anónimo"); };
      MyEvent += (x, y) =>
                 {Console.WriteLine("Da Expressao Lambda: " + x + y); };
      MyEvent?.Invoke(11,22); ;
    }
    public static void MetodoParaOMyEvent(int x, int y){
      Console.WriteLine("Do método tradicional: " + x + "_" + y);
    }
  }
}
```

1. DEFINIÇÃO: Na classe onde será disparado declara-se o event para um tipo de delegate

2. INSCRIÇÃO: nos métodos da classe, onde apropriado inscrevemos delegates no event

3. Onde apropriado dispara o event se este não for null

29

C# - Evolução da sintaxe de delegates

- Evolução dos Delegates no c#
  - No .NET 1.0 instanciava-se um delegate inicializando-o com um método definido algures no código.
  - No .NET 2.0 instanciava-se um delegate com código "inline" denominado método anónimo
  - No .NET 3.0 instancia-se um delegate com uma expressão lambda
- Código para o .NET 3.5 e posterior deve usar Func e Action

```
class Test {
  delegate void TestDelegate(string s);      //1
  static void M(string s) {                  //2
    Console.WriteLine(s);
  }

  static void Main(string[] args) {
     TestDelegate testDel1 =                 //3
        new TestDelegate(M);                 //4

     TestDelegate testDel2 =                 //2
        delegate(string s) {Console.WriteLine(s);
        };                                   //3

     TestDelegate testDel3 =                 //2
        (x) => Console.WriteLine(x);         //3
     ;

     //Invoke the delegates.
     testDel1("Hello. My name is M");        //5
     testDel2("That's nothing. I'm anonymous ");//4
     testDel3("I'm A Lambda Expression");    // 4
  }
}
```


30

C# - Aula

1. Abra o Visual Studio e abra estes slides
2. Abra o ficheiro S041_CodigoMain.txt
3. Crie um projeto denominado S041_CSharpAvancado_NA
4. i = 3;
4.1 **Enquanto** i < 29
   5. i++;
   5.1. Leia o slide [i] !!!
   5.2 Copie o código do ficheiro S041_CodigoMain.txt correspondente ao slide [i] para o método main (apenas o correspondente ao slide [i])
   5.3. Copie as classes necessárias para dentro do seu projeto (e apenas as necessárias para correr o código do slide [i])
   5.4. Nas classes importadas comente o código desnecessário.
   5.5. Leia os comentários e compreenda o que se pretende ilustrar
   5.6. Corra o código e tente interpretar o que se está a passar
   5.7. Se tiver dúvidas … p e r g u n t e … please!!!
6 . Volte ao ponto 4.1 …

31

Estrutura de uma Classe (Pedagogia)

```
public class Mochila {// CABEÇALHO DA CLASSE

    // 1º CONSTANTES
    private final int MAXIMO_DE_MOCHILAS = 120;

    // 2º ATRIBUTOS DE CLASSE
    private static int qtdMochilas;

    //3º MÉTODOS DE CLASSE
    public static int getQtdMochilas(){return qtdMochilas;}

    // 4º DECLARAÇÕES DE ATRIBUTOS e PROPRIEDADES
    private String cor;

    // 5º CONSTRUTORES da classe
    public Mochila(String cor){ this.cor = cor;}

    // 6º principais MÉTODOS geters / seters / ToString
    public void setCor(String cor) {this.cor = cor;}
    public String getCor() {return cor;}
    public String ToString(){return "Cor: " + cor;}

    // 7º Outros MÉTODOS - Processamento!
}
```

32

Estrutura de uma Classe (Encobrimento)

```
public class Mochila {// CABEÇALHO DA CLASSE

// 1º CONSTANTES
    private final int MAXIMO_DE_MOCHILAS = 120;

// 2º ATRIBUTOS DE CLASSE
    private static int qtdMochilas;

    //3º MÉTODOS DE CLASSE
    public static int getQtdMochilas(){return qtdMochilas;}

// 3º Outros MÉTODOS - Processamento!

// 4º CONSTRUTORES da classe
    public Mochila(String cor){ this.cor = cor;}
// 5º principais MÉTODOS geters / seters / ToString
    public void setCor(String cor) {this.cor = cor;}
    public String getCor() {return cor;}
    public String ToString(){return "Cor: " + cor;}

// 6º DECLARAÇÕES DE ATRIBUTOS
    private String cor;
}
```

33