**puc.png PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS - BARREIRO**

**Programação Orientada a Objetos**

**Prática 01 – Classes e objetos**

**Paula Magalhães Alves**

**Curso: Sistemas de Informação**

**Profa. Nilma Rodrigues Alves**

**2º Semestre/2016**

**Questão 1:**

**1.1**

Classe é uma estrutura que contém um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos.

Exemplos de classes:

* Código 1: a classe é denominada **Lista** e sua estrutura se inicia na linha 3;
* Código 2: a classe é denominada **Program** e sua estrutura se inicia na linha 4.

Objeto é uma abstração de algo do mundo real, podendo ser algo palpável ou não, é também uma instância de uma classe, onde são inicializados os atributos e podemos assim chamar os métodos.

Exemplos de objetos:

* Código 1: o objeto é denominado **elementos** e está sendo instanciado na linha 9;
* Código 2: o objeto é denominado **lista** e está sendo instanciado na linha 9.

Atributo é uma propriedade/característica do objeto, podemos dizer que os atributos são as variáveis/campos. Com elas podemos definir o estado de um objeto podendo alterar seus valores.

Exemplos de atributos:

* Código 1: os atributos estão denominados de **elementos** e **qtdaElem,** estão sendo criados nas linhas 5 e 6 respectivamente;
* Código 2: não possui atributo.

Construtor é um método especial onde tem a definição da classe que permite a criação do objeto e é invocada automaticamente sempre que é instanciado um novo objeto com a palavra reservada new.

Exemplos de construtores:

* Código 1: o construtor está denominado de **Lista** e está sendo criado na linha 7;
* Código 2: não possui construtor.

**1.2**

**Código 1:**

* Namespace é utilizado para organizar o código separando arquivos de classe, neste caso ele está “criando” uma pasta chamada **exemplo1\_ListaEncapsulamento**.

namespace exemplo1\_ListaEncapsulamento

* Utilizando a palavra reservada class, é criado uma classe denominada **Lista.**

class Lista

* Criando o atributo denominado **elementos**, que é um vetor de tipo inteiro com visibilidade pública.

public int[] elementos;

* Criando o atributo **qtdaElem**, que é do tipo inteiro com visibilidade pública.

public int qtdaElem;

* Criando um método construtor denominado **Lista**, de visibilidade pública que recebe como parâmetro a variável **tamLista** que é do tipo inteiro.

public Lista(int tamLista)

* Atribuindo a **elementos** uma instância de tipo inteiro de um vetor que recebe como parâmetro a variável denominada **tamLista** que foi recebida como parâmetro no construtor **Lista**.

elementos = new int[tamLista];

* Atribuindo o valor zero a variável denominada **qtdaElem**.

qtdaElem = 0;

**Código 2:**

* Usando a biblioteca denominada **System**, que possui vários métodos já prontos do C # para facilitar os comandos do sistema.

using System;

* Namespace é utilizado para organizar o código separando arquivos de classe, neste caso ele está “criando” uma pasta chamada **exemplo1\_ListaEncapsulamento**.

namespace exemplo1\_ListaEncapsulamento

* Utilizando a palavra reservada class, é criada uma classe denominada **Program.**

class Program

* Método padrão do C # para que o programa inicialize.

static void Main(string[] args)

* Criando uma variável do tipo inteiro denominada **tam** e inicializando ela com valor 7.

int tam = 7;

* Criando uma variável do tipo **Lista** e atribuindo uma instância da classe **Lista** chama o construtor automaticamente pela palavra reservada new e recebe a variável denominada **tam** como parâmetro.

Lista lista = new Lista(tam);

* Utilizando o objeto denominado **lista** e acessando o atributo do tipo vetor na posição zero e atribuindo a ele o valor 1.

lista.elementos[0] = 1;

* Utilizando o objeto denominado **lista** e acessando o atributo denominado **qtdaElem** e incrementando +1.

lista.qtdaElem++;

* Utilizando o objeto denominando **lista** e acessando o atributo do tipo vetor na posição 1 e atribuindo a ele o valor 2.

lista.elementos[1] = 2;

* Utilizando o objeto denominado **lista** e acessando o atributo denominado **qdtaElem** e incrementando +1.

lista.qtdaElem++;

* Utilizando o objeto **lista** e acessando o atributo **qtdaElem** para verificar se o seu valor é maior que zero. Se a condição for verdadeira irá executar as linhas 17 a 23.

if (lista.qtdaElem > 0)

* Criando um laço de repetição, inicializando a variável denominada **i** do tipo inteiro com o valor zero, verifica se o valor da posição atual é menor que o valor do atributo do objeto **lista**, e incrementa +1 na atual posição.

for(int i = 0; i < lista.qtdaElem; i++)

* Imprime no console para cada elemento substituindo o que há nas chaves pelos próximos parâmetros do Console.WriteLine().

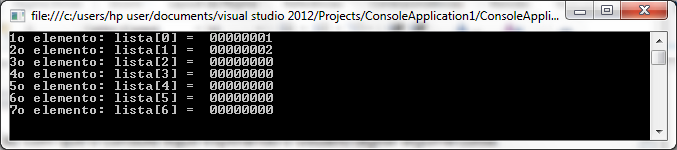
Console.WriteLine("{0}o elemento: lista[{1}] ={2,10:D8}", (i+1), i, lista.elementos[i]);

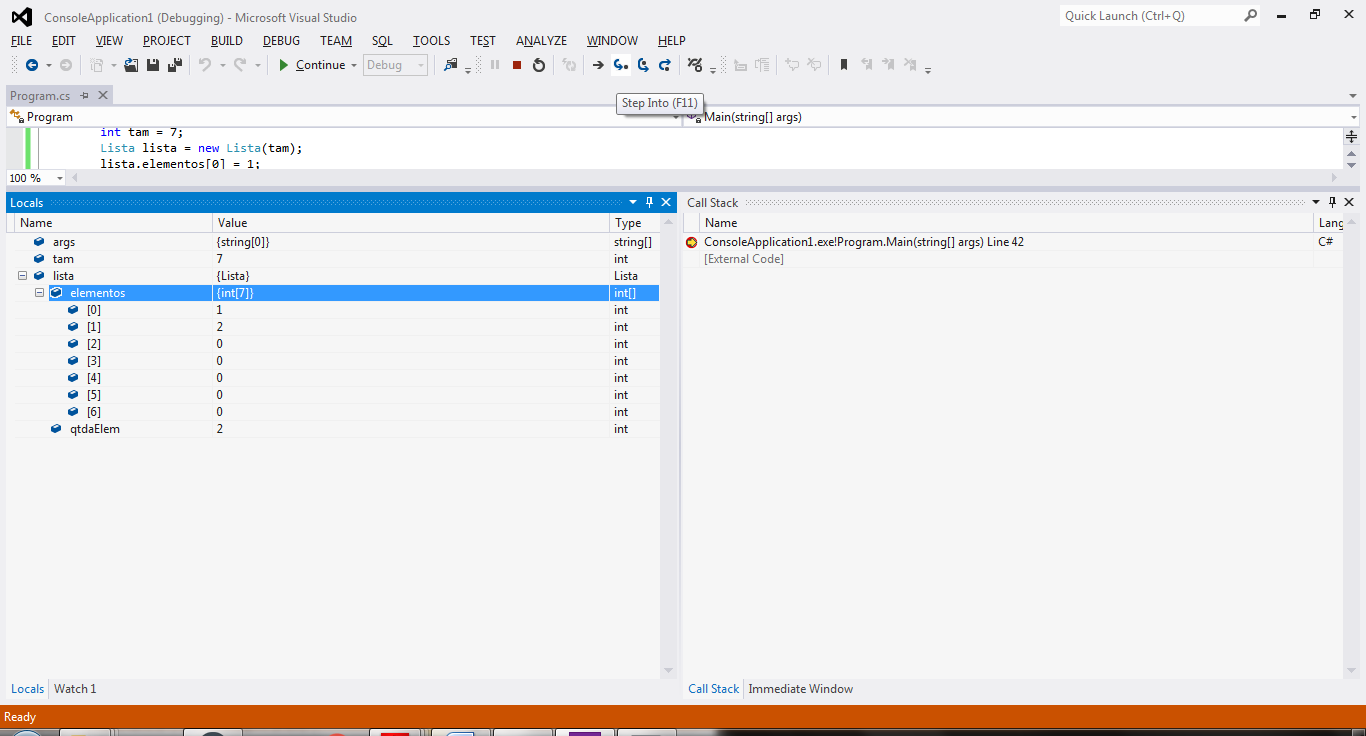
* Faz com que o console fique esperando o usuário digitar alguma coisa.

Console.ReadKey();

**1.3**

A sua saída e estrutura serão respectivamente:





Trocando o código pelo informado há um erro de semântica, pois o código só preenche duas posições do vetor e o laço de repetição está utilizando todas as posições, mostrando valores zerados.

**Questão 2:**

**2.1**

O código 1 não utiliza o encapsulamento pois os seus atributos e/ou métodos estão todos públicos, o que seria uma má prática.

* Visibilidade pública (não encapsulado)

public int[] elementos;

public int qtdaElem;

Como o código 1 não está encapsulado, consequentemente o código 2 utiliza o acesso aos atributos de forma direta, como por exemplo **lista.qtdaElem++**.

* Acesso direto aos atributos (não encapsulado)

lista.elementos[0] = 1;

lista.qtdaElem++;

lista.elementos[1] = 2;

lista.qtdaElem++;

if (lista.qtdaElem > 0)

for(int i = 0; i < lista.qtdaElem; i++)

Console.WriteLine("{0}o elemento: lista[{1}] = {2,10:D8}", (i+1), i, lista.elementos[i]);

O código 3 utiliza do encapsulamento, pois os seus atributos tem visibilidade privada e também possui métodos para retornar e definir valores aos atributos.

* Visibilidade privada (encapsulada)

private int[] elementos;

private int qtdaElem;

* Visibilidade pública pois são os que chamamos de getters e setters (encapsulado)

public int getQtdaElem()

public void add(int elem)

O código 4 não está totalmente encapsulado, pois na linha 21 o objeto **lista** está acessando o atributo **elementos** de forma errada uma vez que o atributo é privado.

* Acesso direto aos setters (encapsulado)

lista.add(1);

lista.add(2);

* Acesso direto aos getters (encapsulado)

qtda = lista.getQtdaElem();

* Acesso ao atributo (não encapsulado)

Console.WriteLine("{0}o elemento: lista[{1}] = {2,10:D8}", (i + 1), i, lista.elementos[i]);

**2.2**

O erro acontece devido na linha 21 do código 4, o objeto **lista** acessa o atributo **elementos** no qual o mesmo é um atributo privado da classe **Lista**.

public int getElemento(int i)

{

if (i > elementos.Length)

{

Console.Write("Posição vetor não existe!");

return -1;

}

return elementos[i];

}

**2.3**

O método **add** na classe **Lista** do código 3 pode apresentar problema semântico, pois se o **add** for chamado mais de 7 vezes irá dar erro devido ao fato do vetor ter apenas 7 posições e não ser validado na criação do método. Uma possível solução seria criar um atributo privado do tipo int para controlar a quantidade de vezes que o método **add** for chamado.

private int qtdAdd = 0;

//Adiciona um elemento na lista

public void add(int elem)

{

if (qtdAdd < elementos.Length) {

elementos[qtdaElem] = elem;

qtdaElem++;

}

qtdAdd++;

}

**2.4**

Esta funcionalidade não está bem modelada, pois tratando de programação orientada a objetos o ideal seria criar um método para impressão dos elementos da lista.

public void imprimeLista()

{

if (this.qtdaElem > 0)

{

for (int i = 0; i < this.qtdaElem; i++)

{

Console.WriteLine("{0}o elemento: lista[{1}] = {2,10:D8}", (i + 1), i, this.getElemento(i));

}

}

}

**Questão 3:**

**3.1**

Não deu erro de compilação, pois o erro é semântico. Mesmo assim será impresso valores padrões de cada tipo de dados. Não é uma boa prática não inicializar valores para as variáveis, pois pode acarretar algum erro durante a execução do programa e na hora de realizar a impressão não ocorrer como o esperado.

**3.2**

É possível porque assim como em outras linguagens o C# possui a possibilidade de não haver um construtor definido no código. Por padrão ao compilar ele gera um construtor vazio.

**Questão 4:**

**4.1**

O nome e o alias saíram vazio, pois como o **this** foi removido do construtor a referência das variáveis são feitas a elas mesmas no construtor, assim o construtor não acessará o atributo da classe por ter o mesmo nome dos parâmetros.

**4.2**

Será passado o objeto **Employee**.