

- Introdução a Programação Orientada a Objetos
- Criação de uma Classe
- Características de uma Classe
- Comportamentos de uma Classe
- Generalização/Especialização
- Herança

Introdução

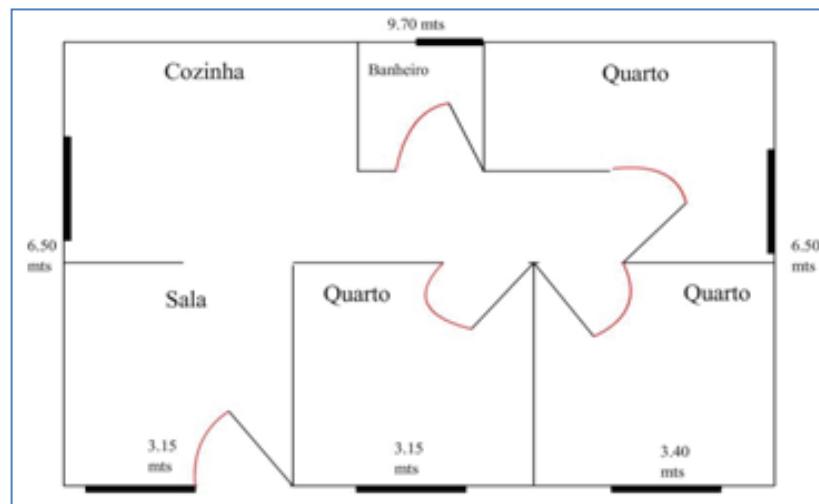
Estamos trabalhando com uma poderosa Linguagem de Programação Orientada a Objetos, porém até o momento demos foco apenas na lógica de programação usando recursos de Orientação à Objetos, porém não nos detalhes deste conceito que é muito importante para o desenvolvimento de aplicativos.

A linguagem de programação C# é orientada a objetos e portanto, alguns conceitos são importantes saber, como por exemplo o que são classes, métodos, construtores, objetos dentre outros que existem na Programação Orientada a Objetos (POO).

Por exemplo, como já comentado em aulas anteriores, a instrução *Console.WriteLine()*; está fazendo uso dos recursos de orientação a objetos, sendo que *Console* é uma classe que contém vários métodos/funções para tratamento de recursos relacionados ao Console do DOS (tela preta) e *WriteLine()* é um método desta classe.

Classes e Objetos

Os termos *classe* e *objeto* algumas vezes são usados alternadamente, mas na verdade, as classes descrevem o tipo de objetos, enquanto objetos são as *instâncias* de classes. Portanto, o ato de criar um objeto é chamado *instanciação*. Usando a analogia da planta de uma casa, a classe é a planta, e o objeto é a construção feita daquela planta.



Classe: Planta



Objeto: Casa

O que são classes?

É uma estrutura de dados que combina estado (campos) e comportamentos (métodos).

Fornecem a definição da estrutura das instâncias (objetos) que serão criadas dinamicamente.

Suportam os mecanismos de herança e polimorfismo, que permitem uma classe derivada estender e especializar uma classe base, estas são propriedades fundamentais do conceito de orientação a objetos.

A declaração de uma classe especifica os atributos e modificadores da classe, o nome da classe, a classe base (se herdar de alguma) e as interfaces implementadas (se existir).

Instâncias da classe, objetos, são criadas usando o operador “new”.

A criação de uma instância aloca memória para o objeto criado, invoca o construtor para inicializar o objeto e retorna a referência para a instância.

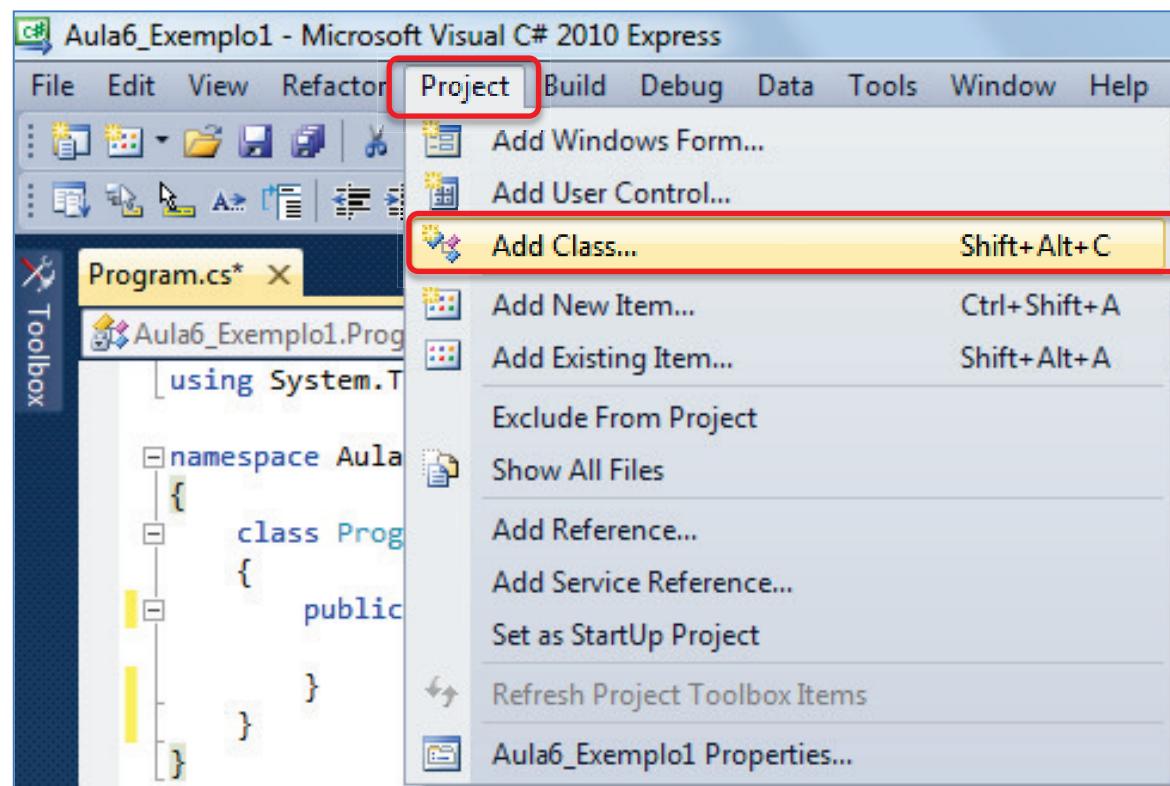
O objeto alocado, quando não mais referenciado, será limpado da memória pelo objeto Garbage Collector.

Criando uma classe

Para criar uma classe em um projeto, basta seguir os seguintes passos:

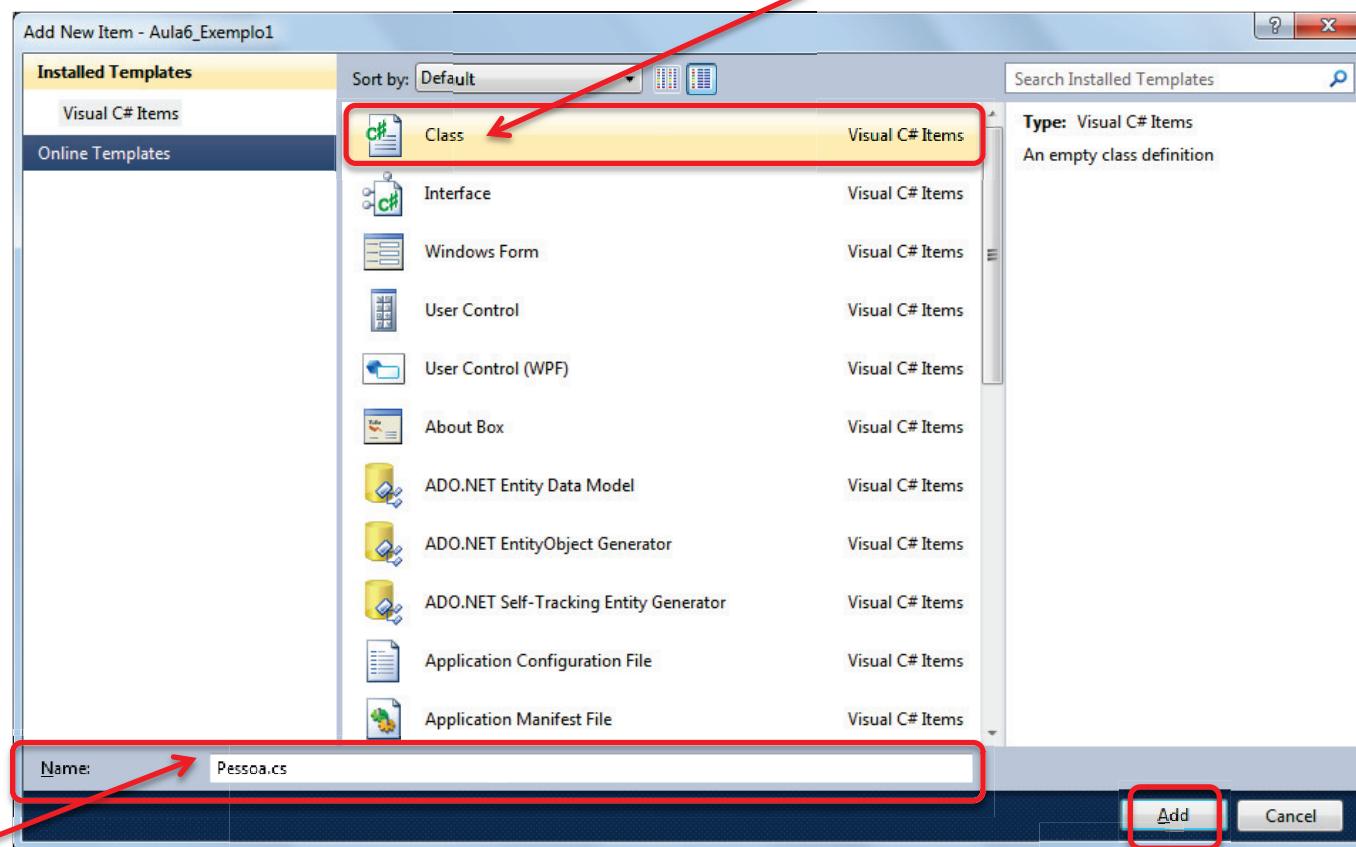
1º) Clicar em *Project*

2º) Clicar em *Add Class...*



Criando uma classe - Continuação

3º) Selecione Class

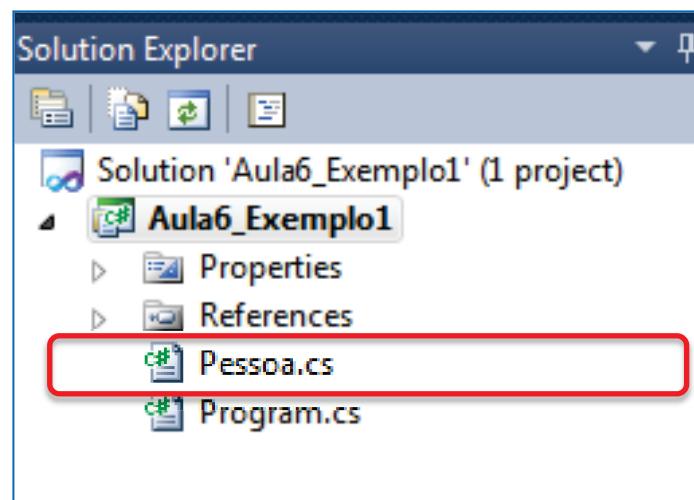


4º) Nome da classe

5º) Clicar em Add

Criando uma classe - Continuação

Depois desses passos, notem que na área do *Solution Explorer*, foi adicionado um arquivo chamado *Pessoa.cs*, que é a classe criada para este exemplo.



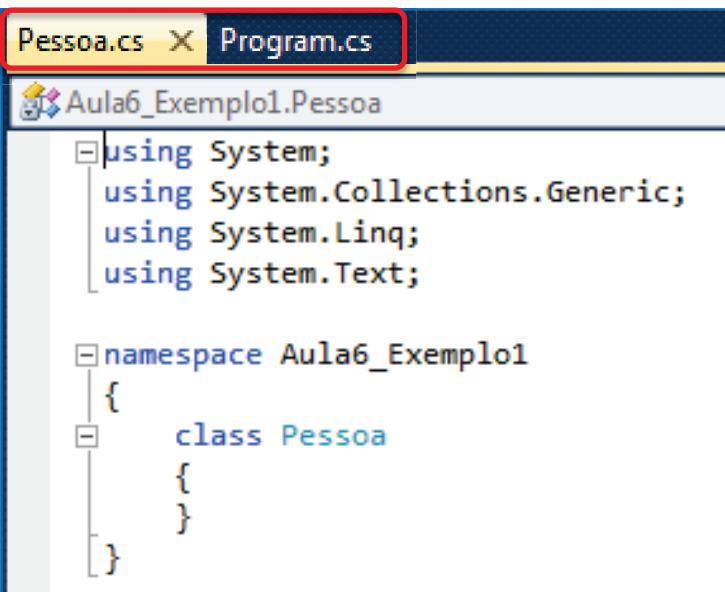
Em Orientação a Objetos o nome do arquivo é o mesmo nome da classe, incluindo a extensão *.cs*, que significa que é uma classe **CSharp**.

Depois basta dar um duplo clique neste arquivo (*Pessoa.cs*) para começar a editá-lo.

Construindo uma classe

Notem que agora existem duas abas na área de edição Program.cs e Pessoa.cs, onde a primeira será a classe principal, onde aplicaremos toda a lógica de programação envolvendo um objeto que será criado do tipo Pessoa.

A segunda aba será a classe Pessoa, onde nela iremos definir seus atributos (campos) e funcionalidades (métodos).



```
Pessoa.cs X Program.cs
Aula6_Exemplo1.Pessoa
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Aula6_Exemplo1
{
    class Pessoa
    {
    }
}
```

Classe Pessoa.cs

```
namespace Aula6_Exemplo1
{
    class Pessoa
    {
        //Atributo da classe
        private string nome;

        //Construtor sem parâmetros
        public Pessoa() {
            this.nome = null;
        }

        //Construtor com parâmetros
        public Pessoa(string nome) {
            this.nome = nome;
        }

        //propriedade do campo nome para acesso ao seu conteúdo
        public string Nome {
            get { return nome; }    //retornar o valor armazenado em um campo.
            set { nome = value; }  //Armazena o valor no campo.
        }

        //Método que retorna uma string contendo as informações da pessoa
        public override string ToString() {
            return (String.Format("Nome: {0}", nome));
        }
    }
}
```

Conceitos de POO

namespace

A palavra-chave de *namespace* é usada para declarar um escopo que contém um conjunto de objetos relacionados. Você pode usar um namespace para organizar elementos de código e para criar globalmente tipos exclusivos.

No exemplo 1, o namespace possui o nome **namespace Aula6_Exemplo1**

Atributos da classe

São campos que representam informações que contém um objeto. Campos são como variáveis.

No exemplo 1, a classe *Pessoa* possui o atributo **private string nome;**

Conceitos de POO

Construtores

Métodos Construtores de instância são utilizados para criar e inicializar instâncias de uma classe. São executados automaticamente quando um objeto de um determinado tipo é criado.

O construtor sem parâmetros é o construtor padrão. Se a classe não define um construtor, o construtor padrão é gerado automaticamente e os campos são inicializados com valores padrão (0, 0.0, null, false).

É declarado da mesma forma que um método, exceto que não tem tipo de retorno (e nem *void*) e deve ter o mesmo nome da classe.

Construtores de instância podem ser sobrecarregados e são invocados com o operador *new*.

Conceitos de POO

Construtores - Continuação

No exemplo 1, temos dois construtores, sendo o primeiro sem parâmetro e outro com parâmetro.

Usado para a criação de um objeto, cujos valores de seus atributos ainda não sabemos no momento de sua criação (instanciação).

Usado para a criação de um objeto, cujos valores de seus atributos já sabemos no momento de sua criação (instanciação).

```
//Construtor sem parâmetros
public Pessoa() {
    this.nome = null;
}

//Construtor com parâmetros
public Pessoa(string nome) {
    this.nome = nome;
}
```

Conceitos de POO

Visibilidade dos membros

Cada membro da classe possui uma visibilidade associada a ele, que controla as áreas de código que podem acessá-lo.

- Public: acesso ilimitado
- Private: acesso limitado a classe.
- Protected: acesso limitado ao this e as classes derivadas.
- Internal: acesso limitado ao assembly.
- Protected internal: acesso limitado ao assembly e as classes derivadas.

Conceitos de POO

Propriedades

Representam uma extensão dos campos. A propriedade tem o mesmo tipo de campo, e a sintaxe para acessá-la é a mesma do campo.

Diferentemente dos campos, as propriedades não representam um local de armazenamento, e possui comando específico para leitura (*get*) e escrita (*set*).

No Exemplo 1, o atributo nome possui sua propriedade chamada Nome. Vou usar esta notação, mas poderia ser qualquer outra, mas que fique claro a sua finalidade.

```
public string Nome {  
    get { return nome; }  
    set { nome = value; }  
}
```

Conceitos de POO

Métodos

É um membro da classe que implementa uma ação que pode ser executada por um objeto ou pela classe.

```
public override string ToString() {
    return (String.Format("Nome: {0}", nome));
}
```

Métodos estáticos são acessados através da classe.

Métodos de instância (não estáticos) são acessados através das instâncias da classe.

Pode ter uma lista de parâmetros, que representa os valores ou referências de variáveis que são passadas para o método; e um tipo de retorno, que indica o que é retornado pelo método.

O método que tem como tipo de retorno void, indica que não retorna um valor.

A assinatura do método deve ser única na classe (o que está dentro do parênteses)

Conceitos de POO

Métodos - Continuação

Parâmetros

São usados para passar valores ou variáveis por referência para os métodos.

Os parâmetros obtém seus valores dos argumentos que são passados para os métodos no momento da execução.

No Exemplo 1, existe um construtor com parâmetro.

```
public Pessoa(string nome) {  
    this.nome = nome;  
}
```

Conceitos de POO

Métodos - Continuação

Sobrecarga

Permite que métodos numa mesma classe tenham o mesmo nome mas assinaturas diferentes.

Quando sobrecarregados, os métodos são localizados pelo compilador de acordo com a sua lista de parâmetros.

Um método específico pode ser selecionado através da conversão dos parâmetros esperados.

No Exemplo 1, existem dois métodos Contrutores com o mesmo nome, porém com assinaturas diferentes, sendo que o primeiro é sem parâmetros e o outro possui parâmetro.

Classe Program.cs

```
namespace Aula6_Exemplo1
{
    class Program
    {
        public static void Main(String[] args) {
            //Criação de um objeto p, do tipo da classe Pessoa
            Pessoa p = new Pessoa();

            //Exibindo na tela informações do objeto p,
            //realizando uma chamada do método ToString
            Console.WriteLine(p.ToString());

            //Atribuindo um nome ao objeto p, por meio da
            //propriedade Nome
            p.Nome = "Tiago";

            //Exibindo na tela informações do objeto p,
            //após a atribuição do nome
            Console.WriteLine(p);

            Console.ReadKey();
        }
    }
}
```

Conceitos de POO

Objetos

Representa uma instância de uma classe.

- Identidade: todo objeto é único e pode ser distinguido de outros objetos.
- Estado: determinado pelos dados contidos no objeto.
- Comportamentos: representados pelos serviços / métodos / operações que o objeto disponibiliza.

Representa algo do mundo real. Exemplo: aluno, cliente, computador, carro.

No Exemplo 1, a criação de um objeto foi realizada pela instrução

```
Pessoa p = new Pessoa();
```

Exemplo 1 – Construtor sem parâmetro - Resultado

Neste exemplo, foi criada uma classe chamada *Pessoa*, que possui suas características e comportamento.

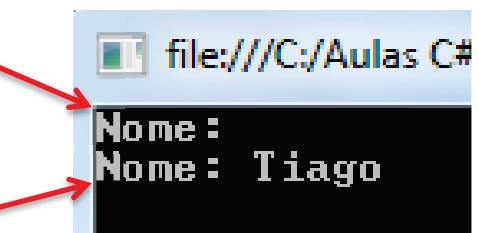
Foi criada também uma classe *Program* que contém o método principal *Main*, que será invocado na execução do programa.

A primeira instrução foi a criação de um objeto chamado *p* do tipo *Pessoa*, e para sua instanciação foi usado o construtor sem parâmetros. Ou seja, ao atributo *nome* do objeto *p* foi atribuído o valor *null* (vazio).

Para demonstrar isso, foi exibido na tela o valor deste atributo.

Na sequência foi atribuído um valor ao atributo *nome*, por meio de sua propriedade *Nome*.

E logo em seguida foi exibido na tela o novo valor do atributo *nome* do objeto *p*.



```
file:///C:/Aulas C#
Nome:
Nome: Tiago
```

Exemplo 2 – Construtor com parâmetro

Neste exemplo, foi usada a mesma classe Pessoa do exemplo anterior.

```
namespace Aula6_Exemplo2
{
    class Program
    {
        static void Main(string[] args)
        {
            //Criação de um objeto p, do tipo da classe Pessoa
            Pessoa p = new Pessoa("Kleber");

            //Exibindo na tela informações do objeto p,
            //realizando uma chamada do método ToString
            Console.WriteLine(p.ToString());

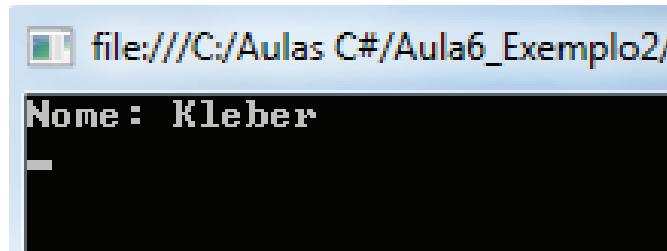
            Console.ReadKey();
        }
    }
}
```

Exemplo 2 – Construtor com parâmetro - Resultado

Neste exemplo, foi criada uma classe chamada *Pessoa*, que possui suas características e comportamento (idem ao exemplo 1).

Foi criada também uma classe *Program* que contém o método principal *Main*, que será invocado na execução do programa (idem ao exemplo 1).

A primeira instrução foi a criação de um objeto chamado *p* do tipo *Pessoa*, mas diferentemente do exemplo anterior, para sua instanciação foi usado o construtor com parâmetro. Ou seja, ao atributo *nome* do objeto *p* foi atribuido o valor “*Kleber*”.



Classe com mais membros

Já que sabemos agora o que são classes e como criar objetos, vamos incrementar mais alguns detalhes na classe *Pessoa*, como por exemplo, os atributos *idade*, *altura* e *sexo*, com suas respectivas propriedades e o método *calcularPesoIdeal*.

Porém para o atributo *sexo* haverá uma regra, onde deverá ser definida uma string padrão sendo “*MASCULINO*” e “*FEMININO*”. Este regra deverá ser definida pela sua propriedade de acesso.

O método *calcularPesoIdeal* deverá retornar o valor do peso ideal, de acordo com a altura.

Exemplo 3 – Classe Pessoa com mais membros

```
namespace Aula6_Exemplo3
{
    class Pessoa
    {
        //Atributos da classe
        private string nome;
        private int idade;
        private double altura;
        private string sexo;

        //Construtor sem parâmetros
        public Pessoa()
        {
            this.nome = null;
            this.idade = 0;
            this.altura = 0.0;
            Sexo = null;
        }

        //Construtor com parâmetros
        public Pessoa(string nome, int idade, double altura, char sexo)
        {
            this.nome = nome;
            this.idade = idade;
            this.altura = altura;
            Sexo = sexo.ToString();
        }
    }
}
```

Exemplo 3 – Classe Pessoa com mais membros - Continuação

```
//propriedade do campo nome para acesso ao seu conteúdo
public string Nome
{
    get { return nome; } //retornar o valor armazenado no campo nome.
    set { nome = value; } //Armazena o valor no campo.
}

public int Idade
{
    get { return idade; }
    set { idade = value; }
}

public double Altura
{
    get { return altura; }
    set { altura = value; }
}

public string Sexo
{
    get { return sexo; } //retornar o valor armazenado no campo sexo.
    set { sexo = value.ToUpper() == "M" ? "MASCULINO": "FEMININO"; }
}
```

Exemplo 3 – Classe Pessoa com mais membros – Continuação

```
//método para calcular o peso ideal de uma pessoa
public double calcularPesoIdeal()
{
    if (sexo.Equals("MASCULINO"))
        return (72.7 * altura) - 58;
    else
        return (62.1 * altura) - 44.7;
}

//Método que retorna uma string contendo as informações da pessoa
public override string ToString()
{
    return (String.Format(" Nome: {0} \n Idade: {1} \n Altura: {2} \n Sexo: {3}\n",
        nome, idade, altura, sexo));
}
```

Para qualquer que seja o sexo da pessoa, será calculado o peso ideal correspondente.

Exemplo 3 – Classe Program

```
namespace Aula6_Exemplo3
{
    class Program
    {
        static void Main(string[] args)
        {
            //Entrada de dados usando variáveis auxiliares
            Console.Write("Digite o nome da pessoa: "); string n = Console.ReadLine();
            Console.Write("Digite a idade de {0}: ", n); int i = int.Parse(Console.ReadLine());
            Console.Write("Digite a altura de {0}: ", n); double a = double.Parse(Console.ReadLine());
            Console.Write("Digite a letra correspondente ao sexo (F/M): ");
            char s = char.Parse(Console.ReadLine());

            //Criando um objeto, usando o construtor com parâmetros
            Pessoa pessoa = new Pessoa(n, i, a, s);

            //Exibindo os dados da pessoa cadastrada
            Console.WriteLine("\n\n*** Dados da Pessoa ***\n");
            Console.WriteLine(pessoa.ToString());

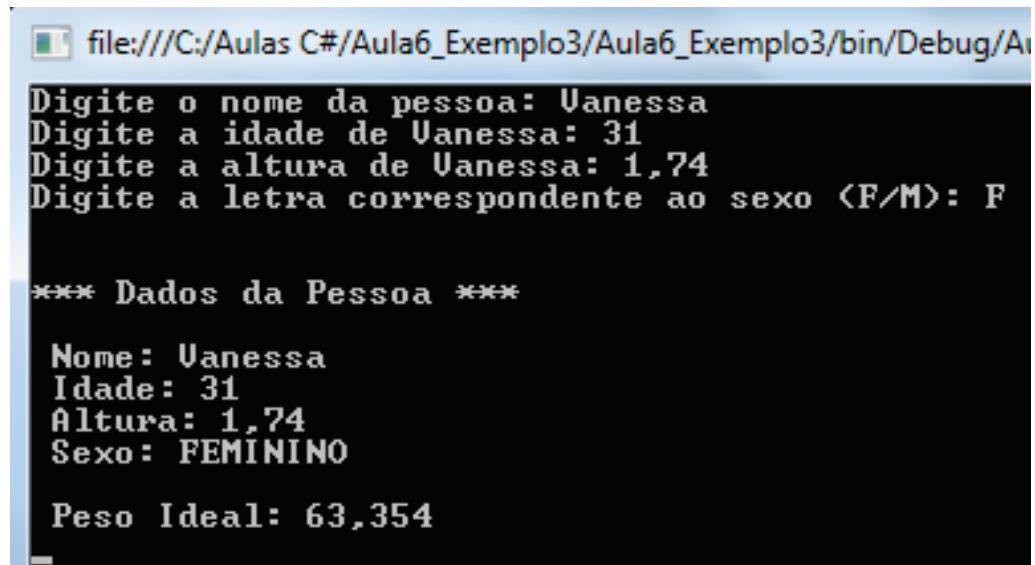
            //Exibindo o resultado do calculo do peso ideal
            Console.WriteLine(" Peso Ideal: {0}", pessoa.calcularPesoIdeal());

            Console.ReadKey();
        }
    }
}
```

Exemplo 3 – Classe Program - Resultado

Neste exemplo podemos observar que quando criamos o objeto pessoa, passamos 4 argumentos como parâmetro, sendo que os mesmos devem estar na mesma ordem de como está definido no método construtor da classe Pessoa, ou seja, deve ser a mesma assinatura.

Não necessariamente o resultado do peso ideal de uma pessoa deve ser exibido na tela como padrão, por isso não foi colocado no método `ToString`, e assim foi possível mostrar que podemos chamar um método a qualquer momento.



```
file:///C:/Aulas C#/Aula6_Exemplo3/Aula6_Exemplo3/bin/Debug/A
Digite o nome da pessoa: Vanessa
Digite a idade de Vanessa: 31
Digite a altura de Vanessa: 1,74
Digite a letra correspondente ao sexo (F/M): F

*** Dados da Pessoa ***
Nome: Vanessa
Idade: 31
Altura: 1,74
Sexo: FEMININO

Peso Ideal: 63,354
```

Arrays (Vetor e Matriz)

Já vimos em aulas anteriores, mas é bom relembrar.

É uma seqüência de elementos, onde todos os elementos são do mesmo tipo.

Cada elemento do array é acesso através de um número inteiro que representa o índice.

Dimensões

`long [] linha;`

- Uma dimensão. Um índice associado a cada elemento `long` do array.

`int [,] matriz;`

- Duas dimensões. Dois índices associados a cada elemento `int`.

Array de Objetos - Vetor

Podemos também criar um array de objetos, ou seja, com base em uma classe podemos criar um vetor, onde neste vetor cada posição dele, será uma instância da classe (objeto).

Por exemplo, a declaração a seguir está reservando um espaço na memória para 50 objetos do tipo Pessoa.

```
Pessoa[] p = new Pessoa[50];
```

Mas para realmente criar os objetos, cada posição deve ser instanciada.

```
p[0] = new Pessoa();
```

```
p[1] = new Pessoa("Juca", 76, 1.57, 'M');
```

```
P[2] = new Pessoa();
```

Array de Objetos - Matriz

Além de arrays bidimensionais de objetos, podemos também trabalhar com arrays multidimensionais.

Por exemplo, a declaração a seguir está reservando um espaço na memória para 60 objetos do tipo Pessoa, estando distribuidos em 15 linhas e 4 colunas.

```
Pessoa[,] p = new Pessoa[15, 4];
```

Mas para realmente criar os objetos, cada posição deve ser instanciada.

```
p[0,0] = new Pessoa();
```

```
p[0,1] = new Pessoa("Manoel", 89, 1.70, 'M');
```

```
P[1,2] = new Pessoa();
```

Exemplo 4 – Array de Pessoas – Método *Main*

Para este exemplo utilizaremos como base a classe Pessoa.

Notem que neste método Main, praticamente foram declaradas as variáveis e estão sendo invocados vários métodos, como *menu*, *cadastrarPessoa*, *listarPessoas* e *exibirMediaPesoIdeal*.

```
namespace Aula6_Exemplo4
{
    class Program
    {
        static void Main(string[] args) {
            const int TAM = 15;
            Pessoa[] p = new Pessoa[TAM]; //Declaração de um vetor de pessoas
            int c = 0; //variável que servirá como um contador de pessoas cadastradas
            int op = 0;
            do {
                op = menu();
                switch (op) {
                    case 1: p[c++] = cadastrarPessoa(); break;
                    case 2: listarPessoas(p, c); break;
                    case 3: exibirMediaPesoIdeal(p, "MASCULINO", c); break;
                    case 4: exibirMediaPesoIdeal(p, "FEMININO", c); break;
                    case 0: break;
                    default: Console.WriteLine("Opção Inválida."); break;
                }
                Console.SetCursorPosition(50, 20);
                Console.Write("Tecle algo para continuar!");
                Console.ReadKey();
            } while (op != 0);
        }
    }
}
```

Exemplo 4 – Array de Pessoas – Método *menu*

Este método será responsável por exibir o menu de opções na tela e solicitar que digite uma das opções.

Logo após o usuário digitar a opção, este método retornará ao método Main um valor inteiro, que será atribuído na variável *op*.

```
public static int menu(){
    Console.Clear();
    Console.WriteLine("**** Menu Principal - Pessoas***\n");
    Console.WriteLine(" 1 - Cadastrar");
    Console.WriteLine(" 2 - Listar todas as pessoas");
    Console.WriteLine(" 3 - Média do Peso Ideal dos Homens");
    Console.WriteLine(" 4 - Média do Peso Ideal das Mulheres");
    Console.WriteLine(" 0 - Sair");
    Console.Write("\n Digite uma opção: ");
    return int.Parse(Console.ReadLine());
}
```

Exemplo 4 – Array de Pessoas – Método *cadastrarPessoa*

Este método será responsável por realizar o cadastro de uma pessoa, solicitando para que o usuário digite as informações da pessoa.

Depois que houver as entradas de dados, na última linha será retornado ao *Main*, um objeto com as características digitadas pelo usuário, que será atribuído em uma posição do vetor *p*, no método principal.

```
public static Pessoa cadastrarPessoa() {
    //Entrada de dados usando variáveis auxiliares
    Console.Clear();
    Console.Write("Digite o nome da pessoa: "); string n = Console.ReadLine();
    Console.Write("Digite a idade de {0}: ", n); int i = int.Parse(Console.ReadLine());
    Console.Write("Digite a altura de {0}: ", n); double a = double.Parse(Console.ReadLine());
    Console.Write("Digite a letra correspondente ao sexo (F/M): ");
    char s = char.Parse(Console.ReadLine());

    //Retornando um objeto, que foi criado usando o construtor com parâmetros
    return new Pessoa(n, i, a, s);
}
```

Exemplo 4 – Array de Pessoas – Método *listarPessoas*

Este método receberá como parâmetros o vetor de objetos, e a quantidade de pessoas cadastradas.

O laço de repetição irá exibir na tela todas as informações de todas as pessoas cadastradas.

```
public static void listarPessoas(Pessoa[] ps, int cp) {
    Console.Clear();
    Console.WriteLine("\n\n*** Dados da Pessoa ***\n");
    for (int x = 0; x < cp; x++)
        Console.WriteLine(ps[x].ToString());
}
```

Exemplo 4 – Array de Pessoas – Método *exibirMediaPesoIdeal*

Este método receberá como parâmetros o vetor de objetos, o sexo e q quantidade de pessoas cadastradas no vetor.

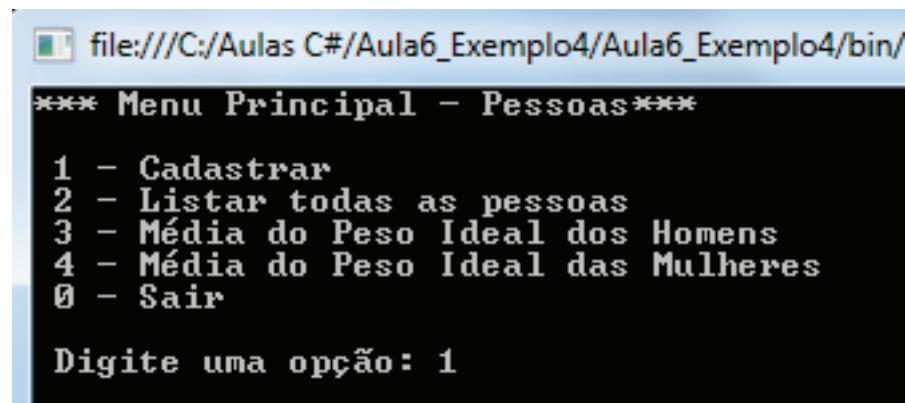
Um laço de repetição servirá para percorrer o vetor, sendo que em cada posição será realizada uma comparação com o sexo da opção escolhida (case 3 ou 4). Sendo verdadeira a condição irá acumular a soma e incrementar no contador. Depois que terminar a verificação, será calculada e exibida a média do peso ideal do referido sexo.

```
public static void exibirMediaPesoIdeal(Pessoa[] p, string sx, int t){  
    double soma = 0; //variável criada para acumular a soma dos pesos ideais  
    int cont = 0; //variável criada para contar a qtde de pessoas do sexo sx  
    for (int x = 0; x < t; x++)  
    {  
        if ((p[x].Sexo).Equals(sx))  
        {  
            soma += p[x].calcularPesoIdeal();  
            cont++;  
        }  
    }  
    Console.WriteLine("Média do peso ideal - {0}: {1} ", sx, soma/cont);  
}
```

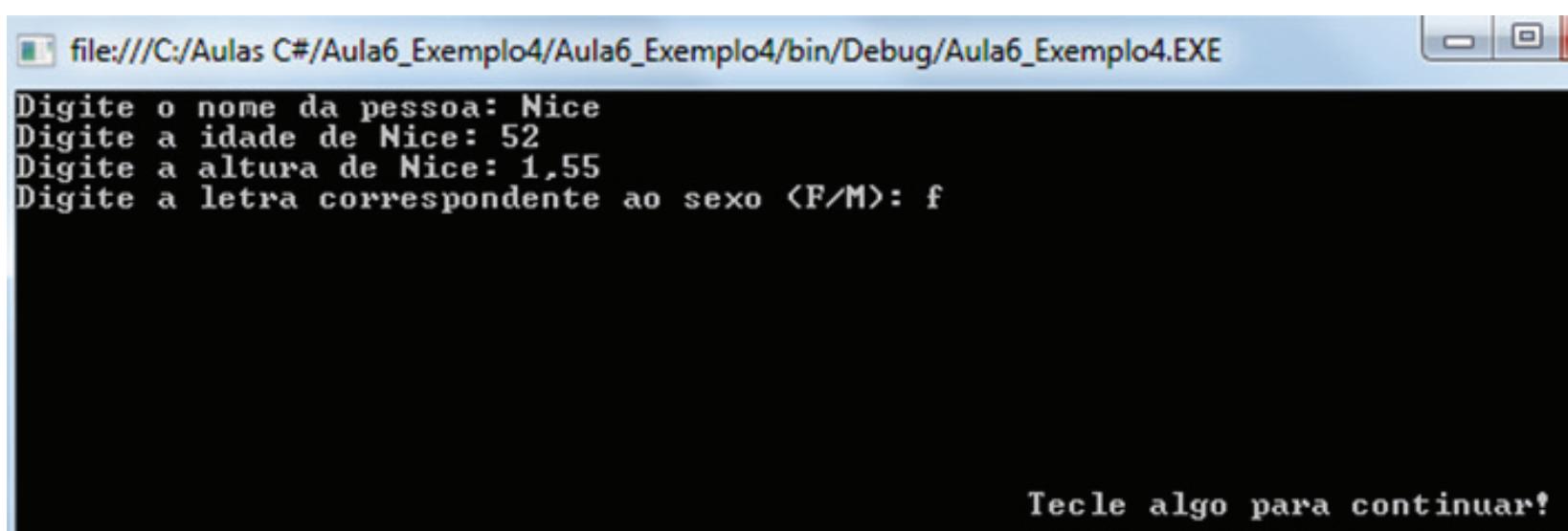
Exemplo 4 – Array de Pessoas – Resultado

Ao executar aparecerá o menu de opções.

Primeiramente realizei 4 cadastros de pessoas para depois testar as outras opções.



```
file:///C:/Aulas C#/Aula6_Exemplo4/Aula6_Exemplo4/bin/  
*** Menu Principal - Pessoas ***  
1 - Cadastrar  
2 - Listar todas as pessoas  
3 - Média do Peso Ideal dos Homens  
4 - Média do Peso Ideal das Mulheres  
0 - Sair  
Digite uma opção: 1
```



```
file:///C:/Aulas C#/Aula6_Exemplo4/Aula6_Exemplo4/bin/Debug/Aula6_Exemplo4.EXE  
Digite o nome da pessoa: Nice  
Digite a idade de Nice: 52  
Digite a altura de Nice: 1,55  
Digite a letra correspondente ao sexo <F/M>: f  
Tecle algo para continuar!
```

Exemplo 4 – Array de Pessoas – Resultado - Continuação

Ao escolher a opção 2, serão listadas todas as informações das pessoas cadastradas.

```
file:///C:/Aulas C#/Aula6_Exemplo4/Aula6_Exemplo4/bin/Debug/Aula6_Exemplo4.exe

*** Dados da Pessoa ***
Nome: Nice
Idade: 52
Altura: 1,55
Sexo: FEMININO

Nome: Alberto
Idade: 57
Altura: 1,65
Sexo: MASCULINO

Nome: Marcia
Idade: 43
Altura: 1,7
Sexo: FEMININO

Nome: Eduardo
Idade: 59
Altura: 1,77
Sexo: MASCULINO
```

Tecle algo para continuar!

```
file:///C:/Aulas C#/Aula6_Exemplo4/Aula6_Exemplo4/bin/Debug/Aula6_Exemplo4.exe

*** Menu Principal - Pessoas ***
1 - Cadastrar
2 - Listar todas as pessoas
3 - Média do Peso Ideal dos Homens
4 - Média do Peso Ideal das Mulheres
0 - Sair

Digite uma opção: 2
```

Exemplo 4 – Array de Pessoas – Resultado - Continuação

Ao escolher a opção 3, será exibida apenas a média do peso ideal das pessoas cujo sexo é masculino. O mesmo vale para opção 4, porém para o sexo feminino.

Para os dois casos, notem que foi usado o mesmo método *exibirMediaPesoIdeal*.



```
file:///C:/Aulas C#/Aula6_Exemplo4/Aula6_Exemplo4/bin/Debug/Aula6_Exemplo4.EXE
*** Menu Principal - Pessoas ***
1 - Cadastrar
2 - Listar todas as pessoas
3 - Média do Peso Ideal dos Homens
4 - Média do Peso Ideal das Mulheres
0 - Sair

Digite uma opção: 3
Média do peso ideal - MASCULINO: 66,317

Tecle algo para continuar!_
```

Exemplo 4 – Array de Pessoas – Observações

Neste exemplo, foram utilizados muito recursos já vistos no curso até o momento, como por exemplo:

- Classes com seus respectivos atributos, propriedades e métodos
- Estruturas de repetição for e do/while
- Estrutura de seleção switch/case
- Estrutura condicional if/else
- Conversões de string para números e caractere
- Métodos com e sem parâmetros
- Métodos com e sem retorno
- Vetor
- Objetos

Mais alguns Fundamentos da Programação Orientada a Objetos

Todas as linguagens gerenciadas no .NET Framework, como por exemplo Visual Basic e C# oferecem suporte completo para programação orientada a objeto, incluindo encapsulamento, herança e polimorfismo.

Encapsulamento: significa que um grupo de propriedades, métodos e outros membros relacionados são tratados como uma única unidade ou um objeto.

Herança: descreve a capacidade de criar novas classes com base em uma classe existente.

Polimorfismo: significa que você pode ter várias classes que podem ser usadas intercambiavelmente, mesmo que cada classe implemente as mesmas propriedades ou métodos de maneiras diferentes.

Herança

Usamos o conceito de herança em POO, quando queremos criar especializações/generalizações de uma classe, como por exemplo, em uma escola temos os seguintes tipos de pessoas envolvidas no cenário, como alunos, professores, funcionários, entre outros.

Cada um desses envolvidos, possuem alguns atributos em comum e outros atributos que são específicos de cada um deles.

Para podermos reutilizar os dados em comum, manteremos a classe Pessoa como base, e criaremos as classes Aluno e Professor, que serão as classes especialistas do nosso cenário para o desenvolvimento.

Herança – Construção de Sub-Classes

Cada classe conterá os seguintes atributos:

Pessoa: nome, idade, altura e sexo

Aluno: matricula e curso

Professor: titulação e carga horária

Em vez de criar as Aluno e Professor com os mesmos atributos repetidamente, podemos reutilizar a classe Pessoa e criar assim a relação entre Aluno e Pessoa, e também entre Professor e Pessoa.

Para evitarmos que neste cenário seja criado um objeto do tipo Pessoa, iremos modificar a classe pessoa como sendo abstrata, ou seja, será uma classe base que não poderá ser criado um objeto com base nela.

A classe Pessoa pode ser usada somente se uma nova classe é derivada dela.

Herança – Representação na forma de Diagrama de Classe

Cada classe conterá os seguintes atributos:

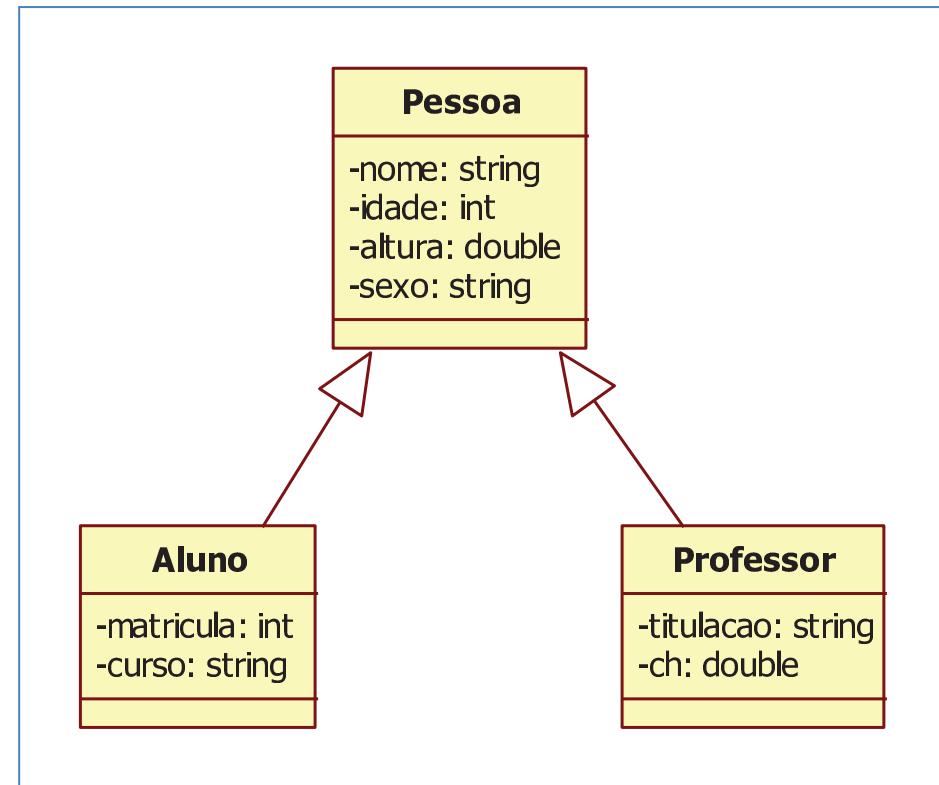
Pessoa: nome, idade, altura e sexo

Aluno: matrícula e curso

Professor: titulação e carga horária

Neste relacionamento podemos dizer que Aluno é uma Pessoa e também Professor é uma Pessoa, com isso ambas herdam as características da classe base.

Podemos também dizer que uma Pessoa pode ser Aluno ou Professor.



Herança – Classe *Pessoa*

Praticamente continua a mesma do exemplo anterior, o que iremos acrescentar é o *abstract* na linha do nome da classe.

```
abstract class Pessoa
{
```

Na classe pessoa, apenas modifiquei o tratamento da propriedade *set* do campo *sexo*, demonstrando outra forma de condição.

```
public string Sexo
{
    get { return sexo; } //retornar o valor armazenado no campo sexo.
    set { sexo = (value == "M" || value == "m") ? "MASCULINO" : "FEMININO"; }
}
```

Herança – Classe *Aluno*

Na classe aluno, temos algumas particularidades, pois temos que associa-la com a classe Pessoa, e para isso usaremos a seguinte notação na linha da classe:

Isso indica que *Aluno* é uma sub-classe de *Pessoa*,
e consequentemente *Pessoa* é a classe base de *Aluno*.

Na sequencia foram declarados os atributos específicos de *Aluno* e depois é criado o construtor padrão, onde precisamos agora acrescentar **:base()** para que seja invocado o construtor da classe base, que é *Pessoa*, e assim inicializando os valores dos atributos.

```
class Aluno : Pessoa
{
    private int matricula;
    private string curso;

    public Aluno()
        : base()
    {
        matricula = 0;
        curso = null;
    }
}
```

Herança – Classe *Aluno* - Continuação

Criaremos na sequencia o construtor com parâmetros.

Como dito anteriormente, uma *Aluno* é uma *Pessoa*, portanto quando formos criar um objeto *Aluno*, teremos que passar todas as informações necessárias, por isso a assinatura deste método construtor receberá todas as informações necessárias.

A classe *Aluno* só possui dois atributos (*matricula* e *curso*), mas como esta associada a classe base que é *Pessoa*, é invocado o construtor da classe base, passando por parâmetro as informações pertinentes a esta classe (*nome*, *idade*, *altura* e *sexo*).

Sendo assim podemos dizer que *Aluno* herda todas as características de *Pessoa*.

```
public Aluno(string nome, int idade, double altura, char sexo, int matricula, string curso)
    : base(nome, idade, altura, sexo)
{
    this.matricula = matricula;
    this.curso = curso;
}
```

Herança – Classe *Aluno* - Continuação

Temos também que criar as propriedades dos atributos da classe Aluno.

```
public int Matricula
{
    get { return matricula; }
    set { matricula = value; }
}

public string Curso
{
    get { return curso; }
    set { curso = value; }
}
```

Herança – Classe *Aluno* - Continuação

E finalizando esta classe, criaremos o método *ToString*, com uma particularidade.

Quando criarmos um objeto do tipo *Aluno* e mandar exibir todos os dados deste objeto, é necessário trazer para método desta classe as informações da classe base, ou seja, trazer para o método *ToString* da classe *Aluno*, todos os dados que já estão formatados para exibição da classe *Pessoa*, e para isso usaremos *base.ToString()*.

```
public override string ToString()
{
    return String.Format("{0} Matricula: {1} \n Curso: {2}",
                         base.ToString(), matricula, curso);
}
```

Obs.: As mesmas regras aplicadas na classe *Aluno*, valem para a classe *Professor*.

Herança – Classe *Program*

Na classe principal, para demonstração de utilização de herança, foram criados dois objetos, e atribuidos à eles valores para seus respectivos atributos.

E na sequencia foram exibidas as informações de cada objeto.

Notem que foi utilizado para a criação o construtor padrão (sem parâmetros) e depois foi sendo atribuído valor para cada campo do objeto.

```
static void Main(string[] args)
{
    Aluno a = new Aluno();
    a.Nome = "Gabrielle";
    a.Idade = 13;
    a.Altura = 1.68;
    a.Sexo = "f";
    a.Matricula = 4231;
    a.Curso = "Turismo";

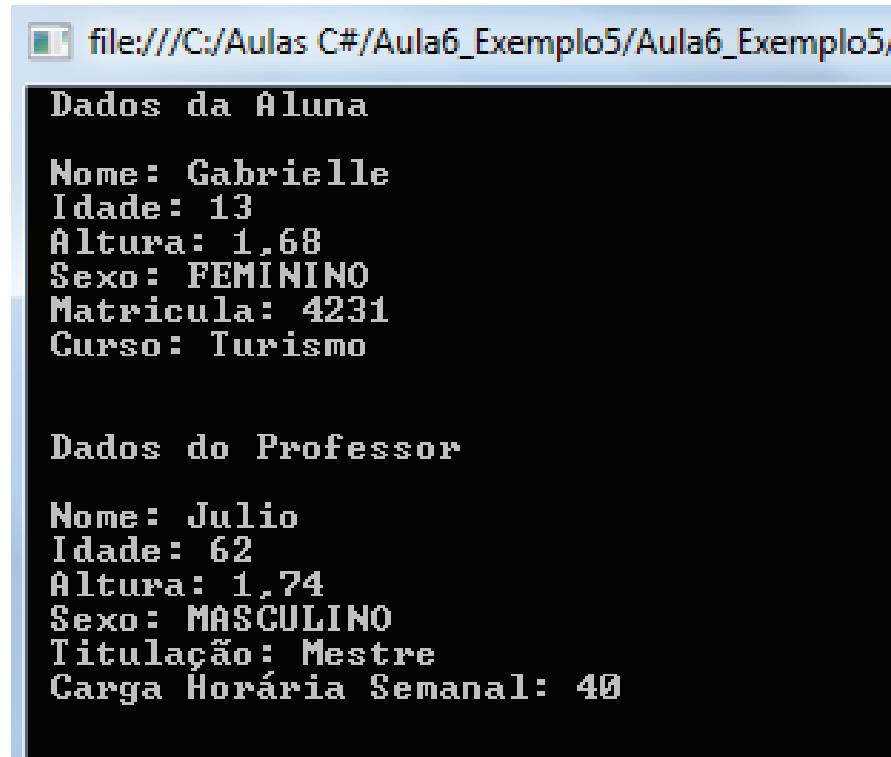
    Console.WriteLine(" Dados da Aluna");
    Console.WriteLine("\n{0}", a.ToString());

    Professor p = new Professor();
    p.Nome = "Julio";
    p.Idade = 62;
    p.Altura = 1.74;
    p.Sexo = "m";
    p.Titulacao = "Mestre";
    p.Ch = 40;

    Console.WriteLine("\n\n Dados do Professor");
    Console.WriteLine("\n{0}", p.ToString());
    Console.ReadKey();
}
```

Herança – Classe *Program* - Resultado

Como resultado temos a seguinte tela:



```
file:///C:/Aulas C#/Aula6_Exemplo5/Aula6_Exemplo5/
Dados da Aluna
Nome: Gabrielle
Idade: 13
Altura: 1,68
Sexo: FEMININO
Matricula: 4231
Curso: Turismo

Dados do Professor
Nome: Julio
Idade: 62
Altura: 1,74
Sexo: MASCULINO
Titulação: Mestre
Carga Horária Semanal: 40
```

Tratamento de Exceções

Normalmente, os aplicativos que possuem diversas tarefas a serem executadas. É possível que uma destas tarefas gere um erro ou exceção.

Por isso devemos dar atenção muito especial à detecção e manipulação de erros em nossas aplicações.

Na Linguagem C#, contamos com um mecanismo que nos auxilia a produzir códigos de manipulação organizados e muito eficientes, que é a manipulação de exceções.

Há vários motivos que podem fazer com que uma exceção seja lançada, tais como uma entrada de valor inválida, cálculo matemático com divisão por zero, falha de hardware, por exemplo.

Tratamento de Exceções – Try/Catch

O manipulador de exceções é um código que captura a exceção lançada e a manipula.

Sua principal vantagem é que precisamos escrever apenas uma vez o código de manipulação de uma exceção que pode ocorrer em uma região controlada.

O bloco de código onde as exceções podem ocorrer é chamado de região protegida, que é indicado pelo comando **try**.

Para associarmos uma determinada exceção a um bloco de código que a manipulará, usamos uma ou mais cláusulas **catch**.

Exemplo 6 - Try/Catch

Ocorrendo algum erro no bloco *try*, a execução será interrompida e srá executado bloco *catch*.

```
static void Main(string[] args)
{
    try {
        Console.Write("Digite um número: ");

        int numero = Convert.ToInt32(Console.ReadLine());

        int resultado = 100 / numero;

        Console.WriteLine("Resultado de 100 / {0} = {1}", numero, resultado );
    }
    catch {
        Console.WriteLine("ERRO");
    }

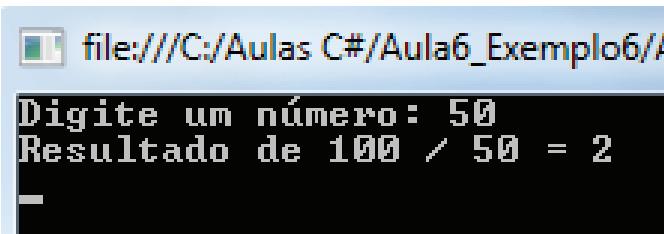
    Console.ReadKey();
}
```

Exemplo 6 - Try/Catch - Resultado

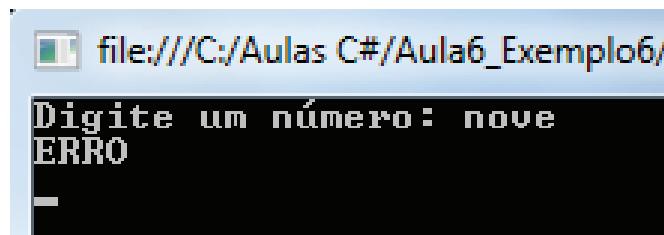
Neste exemplo se o usuário digitar um valor válido, não ocorrerá nenhum erro.

Mas se neste mesmo exemplo, o usuário digitar uma string ao invéz de número, isso gerará erro.

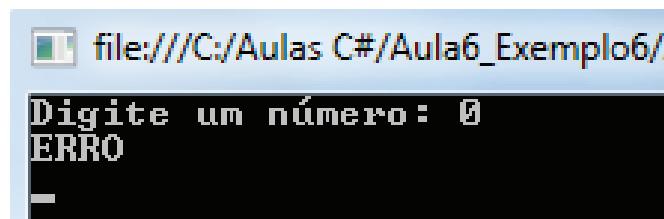
Ou ainda se o número digitado for 0 (zero), também gerará erro.



```
file:///C:/Aulas C#/Aula6_Exemplo6/A
Digite um número: 50
Resultado de 100 / 50 = 2
```



```
file:///C:/Aulas C#/Aula6_Exemplo6/A
Digite um número: nove
ERRO
```



```
file:///C:/Aulas C#/Aula6_Exemplo6/A
Digite um número: 0
ERRO
```

Tratamentos de erros mais específicos

Nas execuções do exemplo anterior, como visto, podem ocorrer diversos tipos de erros.

Sendo assim é possível realizar um tratamento mais refinado do erro o corrido e informar ao usuário o que realmente aconteceu de errado.

Para isso podemos utilizar vários catch's, sendo que cada um irá tratar cada tipo de erro, ficando assim mais fácil de identificar e tratar essas exceções.

Para tratar especificamente estes erros, utilizaremos algumas classes já prontas, fornecidas pelo C# e assim realizar estes tratamentos que são elas:

- *FormatException* – Que tratará de formato de número inválido
- *DivideByZeroException* – Que tratará de divisão por zero
- *Exception* - Que tratará de algum outro tipo de erro que vier a ocorrer. Esta seria a classe Base (Herança) e as anteriores seriam as sub-classes desta.

Exemplo 7 - Try/Catch – Refinado

```
static void Main(string[] args)
{
    try {
        Console.Write("Digite um número: ");
        int numero = Convert.ToInt32(Console.ReadLine());

        int resultado = 100 / numero;

        Console.WriteLine("Resultado de 100 / {0} = {1}", numero, resultado);
    }
    catch (FormatException) {
        Console.WriteLine("Formato de número inválido!");
    }
    catch (DivideByZeroException) {
        Console.WriteLine("O número não pode ser zero!");
    }
    catch (Exception e) { //exceção para caso ocorrer qualquer outro tipo de erro
        Console.WriteLine ("Erro: {0}",e.Message);
    }
    Console.ReadKey();
}
```

Exemplo 7 - Try/Catch – Refinado - Resultado

Neste exemplo se o usuário digitar um valor válido, não ocorrerá nenhum erro.

Mas se neste mesmo exemplo, o usuário digitar uma string ao invéz de número, isso gerará erro.

Ou ainda se o número digitado for 0 (zero), também gerará erro.

```
file:///C:/Aulas C#/Aula6_Exemplo7/
Digite um número: 7
Resultado de 100 / 7 = 14
```

```
file:///C:/Aulas C#/Aula6_Exemplo7/Aula6_Exemplo7.cs:10:1: Error: Invalid number format!
```

```
file:///C:/Aulas C#/Aula6_Exemplo7/A  
Digite um número: 0  
O número não pode ser zero!  
-
```

Considerações

Neste material foi abordado apenas uma introdução básica ao conceito da Programação Orientada a Objetos

Existem muitos conceitos e particularidades que devem ser estudadas mais profundamente, para poder utilizar ao máximo os recursos que este paradigma de linguagem de programação proporciona aos desenvolvedores.

Bibliografia

- MSDN, Microsoft. **Guia de Programação C#**. Disponível:
< http://msdn.microsoft.com/pt-br/library/vstudio/dd460654.aspx >. Acesso em 26 abr 2013
- *< http://andrielleazevedo.wordpress.com/2011/08/11/conceitos-basicos-de-poo-programacao-orientada-a-objetos-para-c-parte-1 / >. Acesso em 27 abr 2013*
- *< http://msdn.microsoft.com/pt-br/library/vstudio/w86s7x04.aspx >. Acesso em 27 abr 2013*
- *< http://msdn.microsoft.com/pt-br/library/bb384054(v=vs.90).aspx >. Acesso em 22 abr 2013*
- *< http://msdn.microsoft.com/pt-br/library/0yd65esw(v=vs.90).aspx >. Acesso em 28 abr 2013*
- *< http://www.devmedia.com.br/try-catch-e-exemplos-tratamento-de-erros-parte-1/18433 >. Acesso em 28 abr 2013*