Programação Orientada a Objetos I

Notas de Aula 01

Prof. Alexandre Mignon

Paradigma

- ➔ Paradigma é uma forma de abordar um problema
- ➔ No contexto da modelagem de um sistema de software, um paradigma corresponde a forma pela qual esse sistema é *entendido* e *construído*.
- ➔ Paradigma Estruturado
  - ◆ Utiliza a técnica de decomposição funcional
  - ◆ Cria um conjunto de procedimentos/funções (algoritmos) para a resolução do problema

Paradigma da Orientação a Objetos

O paradigma da orientação a objetos visualiza um sistema de software como uma coleção de agentes interconectados chamados *objetos*. Cada objeto é responsável por realizar tarefas específicas. É através da interação entre objetos que uma tarefa computacional é realizada.

Paradigma da Orientação a Objetos

A Orientação a Objetos é uma metodologia de desenvolvimento de software em que a principal vantagem é a proximidade com a forma que os seres humanos visualizam e entendem o mundo ao seu redor, tendo como principal objetivo facilitar a modelagem e o desenvolvimento de sistemas por meio da interação entre objetos.

Princípios da Orientação a Objetos

(Alan Kay)

1. Qualquer coisa é um objeto.
2. Objetos realizam tarefas por meio da requisição de serviços a outros objetos.
3. Cada objeto pertence a uma determinada *classe*. Uma classe agrupa objetos similares.
4. A classe é um repositório para comportamento associado ao objeto.
5. Classes são organizadas em hierarquias.

Vantagens da Orientação a Objetos

- ➔ Melhor organização do código do que a empregada nos sistemas não orientados a objetos;
- ➔ Facilita a manutenção e diminui a quantidade de erros de programação (bugs);
- ➔ Maior facilidade para reaproveitamento do código;
- ➔ Maior facilidade para o controle de qualidade e teste de software.

Pilares da Orientação a Objetos

➔ A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da **abstração**, são eles:

- Encapsulamento: é a característica da OO capaz de ocultar partes (dados e detalhes) de implementação interna de classes do mundo exterior;
- Herança: é o mecanismo que permite a uma classe herdar todos os atributos e métodos de outra classe;
- Polimorfismo: é a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras. Ou seja, **várias formas** de responder a mesma mensagem;
- Composição: um objeto mais complexo pode ser composto de partes mais simples.

Abstração

- ➔ Qualquer modelo que inclui os aspectos mais importantes, essenciais de alguma coisa, ao mesmo tempo que ignora os detalhes menos importantes.
- ➔ Permite gerenciar a complexidade e concentrar a atenção nas características essenciais de um objeto.
- ➔ É dependente da perspectiva: o que é importante em um contexto pode não ser importante em outro.

Objetos como Abstrações

- ➔ Uma abstração é uma representação das características e do comportamento relevantes de um conceito do mundo real para um determinado problema.
- ➔ Dependendo do contexto, um mesmo conceito do mundo real pode ser representado por diferentes abstrações.
  - ◆ Carro (para uma transportadora de cargas)
  - ◆ Carro (para uma fábrica de automóveis)
  - ◆ Carro (para um colecionador)
  - ◆ Carro (para um mecânico)

Conceitos OO

➔ Os principais conceitos da OO são:

- Objeto
- Classe
- Mensagem

Classes e Objetos

- ➔ O mundo real é formado por coisas.
- ➔ Na terminologia de orientação a objetos, estas coisas do mundo real são denominadas *objetos*.
- ➔ A descrição de um grupo de objetos é denominada classe de objetos, ou simplesmente de *classe*.

Classe

- ➔ Uma classe é um conjunto de objetos que compartilham uma estrutura comum, um comportamento comum, e uma semântica comum.
- ➔ Pode-se entender uma classe como um **molde** a partir do qual os objetos são construídos.
- ➔ É uma abstração das características relevantes de um grupo de coisas do mundo real.
- ➔ É um elemento de modelagem capaz de fabricar objetos.

Objetos

- ➔ Um objeto é uma instância de uma classe.
- ➔ Um objeto é uma entidade que tem estado, comportamento e uma identidade A estrutura e comportamento de objetos similares são definidos em suas classes comuns.
  - ◆ **Estado**: O estado de um objeto engloba todas as propriedades (geralmente estáticas) de um objeto mais os valores correntes (geralmente dinâmicos) de cada uma dessas propriedades.
  - ◆ **Comportamento**: É como um objeto age e reage, em termos de mudanças do seu estado e passagem de mensagens.
  - ◆ **Identidade**: É aquela propriedade de um objeto que o distingue de todos os outros objetos.

Objetos

- Os objetos possuem:
  - **Características** pelas quais os identificamos;
  - **Finalidades** para as quais os utilizamos
- Características são tipicamente chamadas de **atributos**
  - O objeto Pessoa possui RG, nome, data de nascimento, etc.
  - O objeto Carro possui tipo, cor, quantidade de portas.
- Objetos podem também ter **comportamentos** associados
  - São chamados de **métodos**
  - Um objeto do tipo Pessoa pode andar, correr ou dirigir carros.
  - Um objeto do tipo Carro pode ligar, desligar, acelerar, frear.

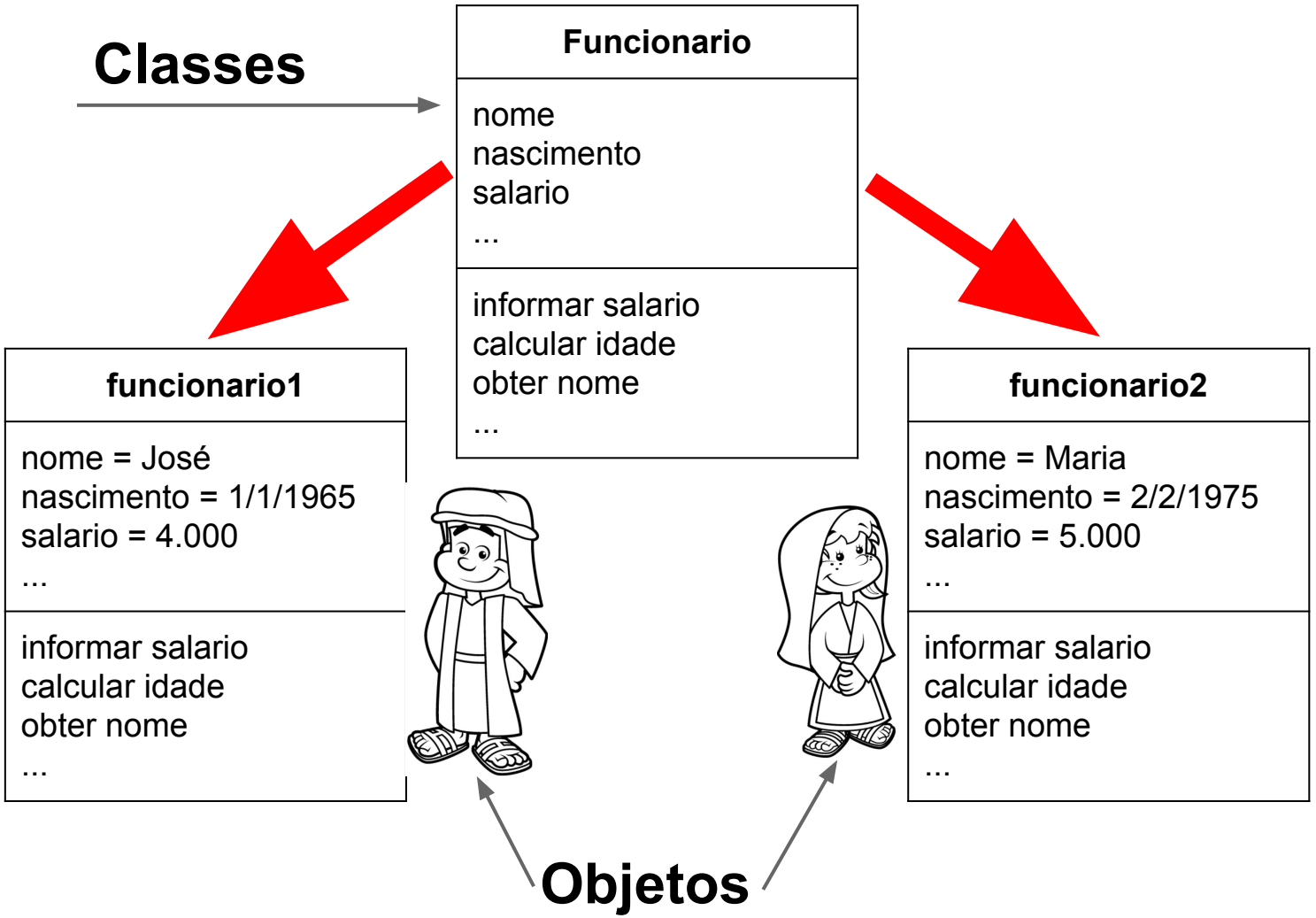
Objetos

- Na POO os objetos possuem **características e comportamentos**
  - As características são chamadas de dados ou **atributos**.
  - Os comportamentos são chamados de operações ou **métodos**.
- Operação x Método
  - **Operação**: elemento de modelagem que representa um serviço que pode ser executado por um objeto.
  - **Método**: implementação de uma operação. Uma possível descrição da execução de uma operação.

Mensagens

- ➔ Para que um objeto realize alguma tarefa, deve haver um estímulo enviado a este objeto.
- ➔ Uma mensagem é uma requisição enviada de um objeto a outro para que este último realize alguma operação.
- ➔ Uma mensagem é o acionamento de um método de um objeto

Classes e Objetos



Exemplo - Classe em C#

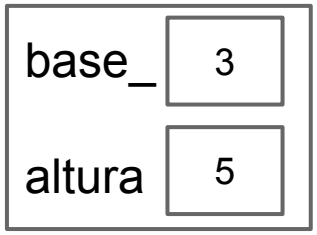
```
namespace Exemplo {
    public class Retangulo
    {
        // atributos
        public float base_;
        public float altura;
        // métodos
        public float CalcularArea()
        {
            return base_ * altura;
        }
    }
}
```

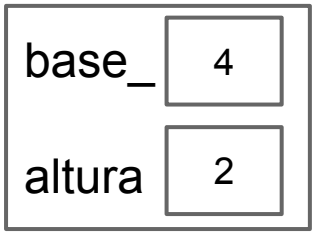
Classes x Objetos

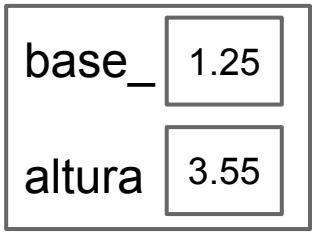
- A classe Retangulo é um modelo

```
public class Retangulo
{
  // atributos
  public float base_, altura;
  // métodos
  ...
}
```

- Objetos do tipo Retangulo são construídos de acordo com o modelo







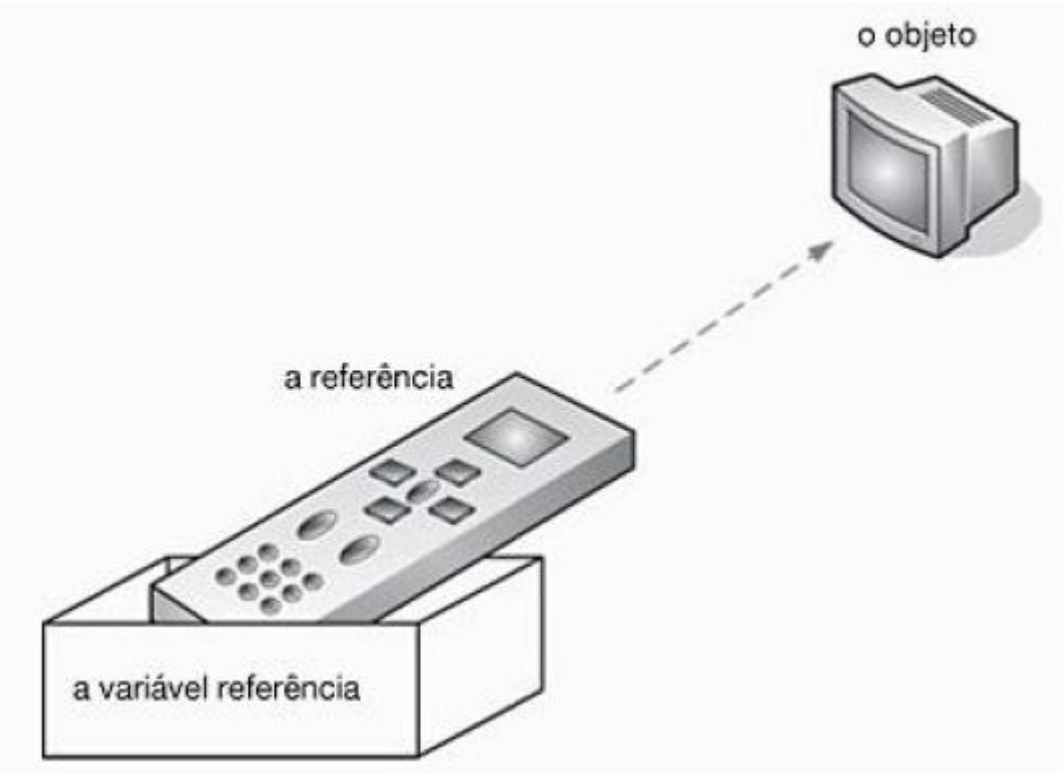
Criando Objetos em C#

- Para criar um objeto em C#, deve-se executar dois passos:
  - **Declarar uma variável para referenciar o objeto**: assim como os tipos primitivos, os objetos devem ser declarados;
  - **Instanciar o objeto**: alocar o objeto em memória. Para isso utilizamos o operador **new** e um método especial conhecido como **construtor**.
- Exemplo

```
Retangulo r1; // declara a variável
r1 = new Retangulo(); // instancia o objeto
```

Criando Objetos em C#

- ➔ O operador **new** retorna o *endereço de memória* do objeto criado.
- ➔ Esse endereço é normalmente atribuído a uma variável **referência**.
- ➔ Uma variável referência pode ser entendida como sendo um "ponteiro" para o objeto.



Criando Objetos em C#

➔ Podemos declarar e instanciar o objeto em uma única linha

```
Retangulo r1 = new Retangulo();
```

➔ Uma vez criado o objeto, podemos acessar os seus atributos usando a notação ponto "."

```
r1.base_ = 3;
r1.altura = 5;
```

➔ Podemos também chamar os seus métodos usando a notação ponto

```
float area = r1.CalcularArea();
```

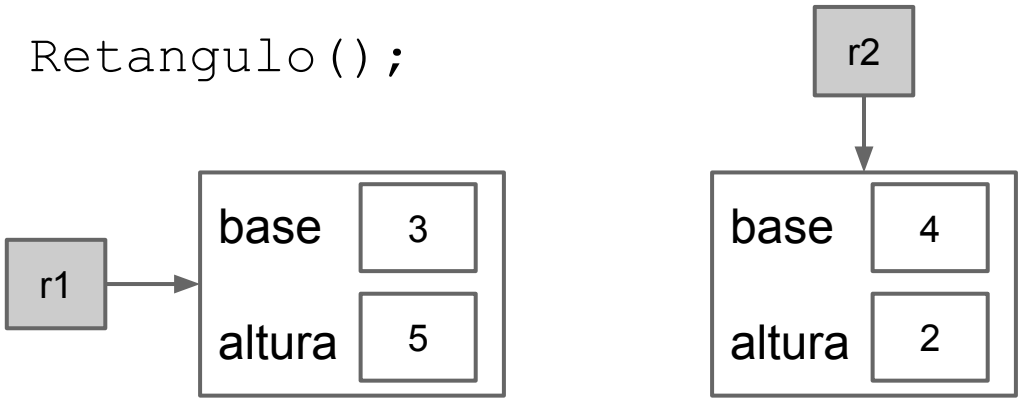
Exemplo - Classe em C#

```
using System;
namespace Exemplo {
    public class Program
    {
        static void Main(string[] args)
        {
            Retangulo r = new Retangulo();
            r.base_ = 3;
            r.altura = 2;
            float area = r.CalcularArea();
            Console.WriteLine("Area: {0}", area);
        }
    }
}
```

Criando Objetos em C#

- Vários objetos podem ser instâncias de uma mesma classe, mas esses objetos são distintos
- Exemplo

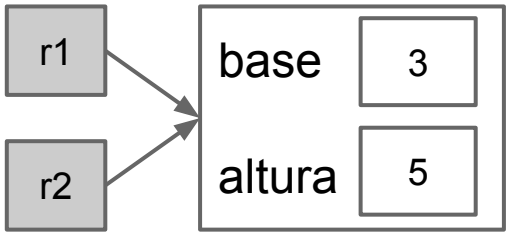
```
Retangulo r1 = new Retangulo();
Retangulo r2 = new Retangulo();
r1.base_ = 3;
r1.altura = 5;
r2.base_ = 4;
r2.altura = 2;
```



Criando Objetos em C#

- Duas ou mais variáveis podem referenciar o mesmo objeto

```
Retangulo r1 = new Retangulo();
r1.base_ = 3;
r1.altura = 5;
Retangulo r2 = r1;
```



Métodos Construtores

- Métodos Construtores são métodos especiais que são chamados automaticamente quando instâncias são criadas através da palavra-chave *new*.
- Por meio da criação de construtores, podemos garantir que o código que eles contêm será executado antes de qualquer outro código em outros métodos.
- Os construtores são úteis para iniciar campos de instâncias de classes para garantir que, quando métodos dessas instâncias forem chamados eles contenham valores específicos.

Métodos Construtores

- ➔ Toda classe deve ter um método construtor.
- ➔ Se nenhum construtor for declarado o compilador da linguagem C# adiciona automaticamente o construtor padrão.
  - ◆ Construtor vazio e sem parâmetro
- ➔ O construtor é um método com mesmo nome que a classe e sem tipo de retorno.
- ➔ Podemos definir vários construtores para uma classe.
  - ◆ Devem ter assinaturas diferentes (parâmetros)
- ➔ Se for declarado algum construtor, o vazio só existe quando declarado
  - ◆ Construtor vazio só não precisa ser declarado se não existe outro construtor.

Métodos Construtores - Exemplo

```
public class Retangulo
{
    // atributos
    public float base_, altura;
    // método construtor - inicializa os atributos com o valor 1
    public Retangulo()
    {
        base_ = 1;
        altura = 1;
    }
 }
```

```
class Program
{
    static void Main(string[] args)
    {
        Retangulo r = new Retangulo();
    }
}
```

Métodos Construtores - Exemplo 2

```
public class Retangulo
{
    // atributos
    public float base_, altura;
    // método construtor com dois parâmetros
    public Retangulo(float b, float a)
    {
        base_ = b;
        altura = a;
    }
 }
```

```
class Program
{
    static void Main(string[] args)
    {
        // como só tem um método construtor com parâmetro é necessário passá-los
        Retangulo r = new Retangulo(3.0f, 4.0f);
    }
}
```

Métodos Construtores - Exemplo 2

```
public class Retangulo
{
    // atributos
    public float base_, altura;
    // método construtor - inicializa os atributos com o valor 1
    public Retangulo()
    {
        base_ = 1;
        altura = 1;
    }
    // método construtor - inicializar os atributos com os valores dos parâmetros
    public Retangulo(float b, float a)
    {
        base_ = b;
        altura = a;
    }
}
```

Métodos Construtores - Exemplo 3

```
class Program
{
    static void Main(string[] args)
    {
        // chama o construtor sem parâmetros
        Retangulo r = new Retangulo();
        // chama o construtor com dois parâmetros do tipo float
        Retangulo r1 = new Retangulo(3.0f, 4.0f);
    }
}
```

A Palavra-Chave *this*

- ➔ É criada internamente para cada instância, uma "auto referência", ou seja, uma referência à própria instância.
- ➔ Essa referência é representada pela palavra-chave *this*.
- ➔ O uso mais imediato é como uma auto referência dos atributos dentro do objeto.
  - ◆ Nele a palavra chave passa a se referir diretamente aos atributos da classe.
- ➔ Ao ser usada dentro de métodos, permite diferenciar entre um atributo e um parâmetro que tem o mesmo nome (identificador).

Exemplo

```
public class Retangulo
{
    // atributos
    public float base_, altura;
    // método construtor com o uso de this
    public Retangulo(float base_, float altura)
    {
        this.base_ = base_;
        this.altura = altura;
    }
}
```

Modificadores de Acesso

➔ Permite restringir o acesso a atributos e métodos das classes.

Modificador	Significado
public	O atributo ou método declarado com este modificador poderá ser acessado ou executado a partir de qualquer outra classe, ou seja sem restrição.
private	O atributo ou método declarado com este modificador só pode ser acessado ou executados por métodos da própria classe, sendo completamente oculto para o programador usuário que usar instâncias desta classe ou criar classes herdeiras ou derivadas.
protected	Funciona como o modificador *private*, exceto que classes herdeiras ou derivadas também terão acesso ao campo ou método declarado com este modificador. Assim é permitido o acesso de todas as classes derivadas.

Regras para o uso de Modificadores

- ➔ Todos os atributos de uma classe devem ser declarados com o modificador *private* ou *protected*, ficando dessa forma ocultos para o programador usuário dessas classes.
- ➔ Métodos que devem ser acessíveis são declarados explicitamente com o modificador *public*.
- ➔ Como a princípio os campos terão o modificador *private*, métodos que permitam a manipulação controlada dos valores dos campos (conhecidos por métodos “getters” e “setters” ou encapsulamento) devem ser escritos nas classes e estes métodos devem ter o modificador *public*.

Encapsulamento

- ➔ Encapsulamento é a característica da OO de ocultar partes independentes da implementação.
- ➔ É também uma forma de **restringir o acesso** ao comportamento interno de um objeto.
  - ◆ Um objeto que precise da colaboração de outro para realizar alguma tarefa simplesmente envia uma mensagem a este último.
  - ◆ O método (maneira de fazer) que o objeto requisitado usa para realizar a tarefa não é conhecido dos objetos requisitantes.
- ➔ Ele é o mecanismo utilizado para disponibilizar métodos, protegendo o acesso direto indevido aos atributos de uma instância (objeto).
- ➔ O encapsulamento evita a interferência externa indevida de um objeto sobre os dados de outros objetos a ele referenciados.

Métodos getters e setters

- Como princípio do encapsulamento os atributos devem ficar ocultos.
  - Devem ser declarados com o modificador **private**
- Para permitir a manipulação controlada dos valores dos atributos são criados métodos conhecidos como “getters” e “setters”.
  - Devem ser declarados como o modificador **public**

Métodos getters e setters

Sem encapsular	Encapsulamento

`public Tipo atributo;`	`private Tipo atributo;`
	`public Tipo getAtributo()`
	`{`
	`  return this.atributo;`
	`}`
	`public void setAtributo(Tipo valor)`
	`{`
	`  this.atributo = valor;`
	`}`

`Referencia.atributo=x;`	`Referencia.setAtributo(x);`
`x=Referencia.atributo;`	`x=Referencia.getAtributo();`

Métodos getters e setters - Exemplo

```
namespace ExemploEncapsulamento
{
    public class Pessoa
    {
        // atributo
        private string nome;
        // métodos getters e setters
        public string GetNome()
        {
            return nome;
        }

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

Métodos getters e setters - Exemplo

```
using System;

namespace ExemploEncapsulamento
{
    class Program
    {
        static void Main(string[] args)
        {
            Pessoa p = new Pessoa();
            p.SetNome("Joao");
            string n = p.GetNome();
        }
    }
}
```

Encapsulamento em C#

- ➔ A forma geral aceita em todas as linguagens OO de encapsulamento é pelos métodos dos “getters” e “setters”.
- ➔ A linguagem C# permite outras formas de encapsulamento. Uma delas é encapsular sem transformar em método, mas criando uma variável de passagem.

Encapsulamento em C# - Exemplo

```
namespace ExemploEncapsulamento
{
    public class Pessoa
    {
        // atributo
        private string nome;

        // variável de passagem - Propriedade
        public string Nome
        {
            get { return nome; }
            set { nome = value; }
        }
    }
}
```

Encapsulamento em C# - Exemplo

```
using System;

namespace ExemploEncapsulamento
{
    class Program
    {
        static void Main(string[] args)
        {
            Pessoa p = new Pessoa();
            p.Nome = "Joao";
            string n = p.Nome;
        }
    }
}
```

Modificador *static*

- ➔ É possível definir atributos estáticos para a classe
  - São compartilhados por todas as instâncias dessa classe, isto é, somente um valor será armazenado em um atributo estático, e caso esse valor seja modificado por uma das instâncias da classe, a modificação será refletida em todas as outras instâncias dessa classe.
- ➔ Para declararmos um atributo estático utilizamos o modificador **static**.
  - Deve ser declarado antes do tipo de dado do atributo.
- ➔ Atributos estáticos também são conhecidos como atributos de classes, já que esses atributos podem ser acessados diretamente usando o nome da classe, sem que sejam necessárias a criação de uma instância da classe e uma referência para tal instância.

Modificador *static* - Exemplo

```
namespace ExemploStatic
{
    public class Exemplo
    {
        private static int seq = 0;
        private int nro;
        public int Nro
        {
            get { return nro; }
            set { nro = value; }
        }
        public Exemplo()
        {
            seq = seq + 1;
            nro = seq;
        }
    }
}
```

Modificador *static* - Exemplo

```
using System;

namespace ExemploStatic
{
    class Program
    {
        static void Main(string[] args)
        {
            Exemplo e1 = new Exemplo();
            Console.WriteLine(e1.Nro);

            Exemplo e2 = new Exemplo();
            Console.WriteLine(e2.Nro);

            Exemplo e3 = new Exemplo();
            Console.WriteLine(e3.Nro);
        }
    }
}
```

Métodos Estáticos

- O modificador **static** também pode ser utilizado para a declaração de métodos.
  - Deve ser declarado antes do tipo de retorno do método.
- Métodos estáticos também são conhecidos como métodos de classes, já que esses métodos podem ser acessados diretamente usando o nome da classe, sem que sejam necessárias a criação de uma instância da classe e uma referência para tal instância.
- Os métodos estáticos podem acessar somente os atributos estáticos da classe.

Métodos Estáticos - Exemplo

```
namespace ExemploStatic
{
   public class Exemplo
   {
       private static int seq = 0;
       private int nro;
       public int Nro
       {
           get { return nro; }
           set { nro = value; }
       }
       public Exemplo()
       {
           seq = seq + 1;
           nro = seq;
       }
       // declaração de um método estático
       public static int ProximoNumero()
       {
           return seq + 1;
       }
   }
}
```

Métodos Estáticos - Exemplo

```
using System;

namespace ExemploStatic
{
   class Program
   {
       static void Main(string[] args)
       {
           Exemplo e1 = new Exemplo();
           Console.WriteLine(e1.Nro);

           Exemplo e2 = new Exemplo();
           Console.WriteLine(e2.Nro);

           Exemplo e3 = new Exemplo();
           Console.WriteLine(e3.Nro);

           // chamada ao método estático
           int prox = Exemplo.ProximoNumero();
           Console.WriteLine("Proximo = {0}", prox);
       }
   }
}
```