# Aula 7

Prof. Ricardo Frohlich da Silva

# Encapsulamento

- Encapsulamento é um dos princípios fundamentais da programação orientada a objetos (POO) e é amplamente aplicado em linguagens como C#.
- Ele se refere à prática de ocultar o estado interno de um objeto e permitir o acesso a esse estado apenas por meio de métodos e propriedades específicos.
- Isso fornece um mecanismo de controle sobre como os dados de um objeto são acessados e modificados, resultando em um código mais organizado, seguro e manutenível.

# Encapsulamento

- A principal finalidade do encapsulamento é fornecer um nível de abstração que separa a implementação interna de uma classe da sua interface pública.
- Isso significa que os detalhes de implementação podem ser alterados sem afetar o código que usa a classe, desde que a interface pública permaneça a mesma.
- Além disso, o encapsulamento ajuda a proteger os dados de uma classe, permitindo a aplicação de lógica de validação e restrições ao acesso aos campos internos.

# Encapsulamento

- O encapsulamento em C# envolve principalmente os seguintes conceitos:
  - Modificadores de Acesso
  - Campos (Fields)
  - Propriedades (Properties)

# Encapsulamento

- Modificadores de Acesso
  - Em C#, você pode usar modificadores de acesso, como private, public, protected e internal, para determinar quem pode acessar os membros da classe.
  - Isso ajuda a restringir o acesso direto a campos e métodos internos da classe.

# Encapsulamento

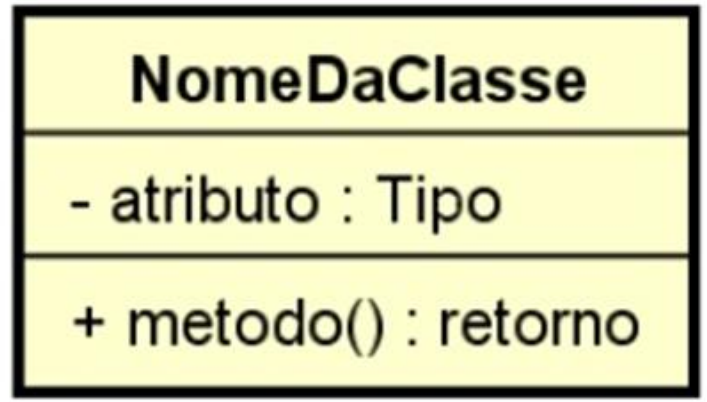
- Modificadores de Acesso
  - private:
    - O modificador private é o mais restritivo em termos de visibilidade.
    - Os membros declarados como privados só podem ser acessados dentro da própria classe onde foram declarados.
    - O objetivo principal do acesso privado é ocultar os detalhes de implementação interna da classe e garantir que os dados sejam protegidos contra acesso direto de fora da classe.
  - public:
    - O modificador public é o mais permissivo em termos de visibilidade.
    - Os membros declarados como públicos podem ser acessados de qualquer lugar no código, dentro ou fora da classe, sem restrições.
    - Os membros públicos são amplamente visíveis e podem ser usados por qualquer parte do código que tenha acesso à classe que os contém.

# Encapsulamento

- Modificadores de Acesso
  - protected:
    - O modificador protected é usado para permitir o acesso a membros dentro da classe onde foram declarados e em classes derivadas (subclasses) da classe que os contém.
    - Isso é útil quando você deseja fornecer acesso controlado aos membros para classes derivadas, mas ainda manter esses membros ocultos para outras classes que não sejam derivadas.
  - internal:
    - O modificador internal limita o acesso a membros para classes dentro do mesmo assembly (conjunto de arquivos compilados).
    - Os membros declarados como internos não podem ser acessados por classes em assemblies externos.
    - O modificador internal é usado para fornecer um nível intermediário de visibilidade, permitindo que membros sejam usados em um conjunto de classes relacionadas no mesmo assembly.

# Encapsulamento – Diagrama de classes

- Atributos
  - <acesso> <nome> : <tipo>
- Métodos
  - <acesso> <nome> (<parâmetros>) : <tipo>
- <acesso> = público: sinal de “+”
- <acesso> = privado: sinal de “-”
- <acesso> = protegido: sinal de “#”



# Encapsulamento

- Campos (Fields)
  - Campos são variáveis de instância privadas que armazenam o estado interno de uma classe.
  - Ao encapsular esses campos com propriedades, você controla como os valores são lidos e gravados, garantindo que a integridade dos dados seja mantida.

# Encapsulamento

- Propriedades (Properties)
  - As propriedades são membros de classe que servem como interfaces para acessar e modificar campos (ou variáveis de instância) privados.
  - Elas geralmente consistem em um getter para recuperar o valor de um campo e, opcionalmente, um setter para modificar o valor.
  - Propriedades permitem a validação e a lógica personalizada ao acessar e definir valores.

# Encapsulamento

- Os getters e setters em C# são métodos especiais usados para acessar e modificar os valores de campos privados de uma classe.
- Eles são uma parte fundamental do conceito de propriedades (properties) e são comumente usados para garantir o encapsulamento e o controle do acesso aos dados.

# Encapsulamento

- Getter (Acessor):
  - Um getter é um método que permite recuperar (obter) o valor de um campo privado.
  - Geralmente, um getter é associado a uma propriedade (property) e é definido com a palavra-chave get.
  - Um getter não tem parâmetros e deve retornar o valor do campo privado correspondente.
  - É responsável por fornecer acesso somente leitura aos dados encapsulados, garantindo que o valor do campo possa ser lido, mas não modificado diretamente.

# Encapsulamento

- Getter (Acessor):

```
public class MinhaClasse
{
    private int _meuCampoPrivado;

    public int MeuCampo
    {
        get
        {
            return _meuCampoPrivado; // Retorna o valor do campo privado
        }
    }
}
```

# Encapsulamento

- Setter (Modificador):
  - Um setter é um método que permite modificar (definir) o valor de um campo privado.
  - É definido com a palavra-chave set e deve ter um parâmetro que especifica o novo valor que se deseja atribuir ao campo.
  - Os setters permitem aplicar validações, lógica personalizada ou regras de negócios ao definir valores em campos privados, garantindo a integridade dos dados.

# Encapsulamento

- Setter (Modificador):

```
public class MinhaClasse
{
    private int meuCampoPrivado;

    public int MeuCampo
    {
        get
        {
            return meuCampoPrivado; // Retorna o valor do campo privado
        }
        set
        {
            if (value >= 0)
            {
                meuCampoPrivado = value; // Define o valor do campo privado
            }
        }
    }
}
```

# Encapsulamento

```
public class Pessoa
{
    private string _nome; // Campo privado para armazenar o nome

    public string Nome
    {
        get { return _nome; } // Getter para acessar o nome
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _nome = value; // Setter para definir o nome
            }
        }
    }
}
```

# Encapsulamento

```
static void Main(string[] args)
{
    Pessoa pessoa = new Pessoa();
    pessoa.Nome = "Ricardo Frohlich da Silva"; // Define o nome usando o setter
    string nomeDaPessoa = pessoa.Nome; // Lê o nome usando o getter
    Console.WriteLine("O nome da pessoa é: "+nomeDaPessoa);
}
```

# Propriedades

- Propriedades em C# são um recurso da linguagem que permite encapsular um campo privado dentro de uma classe ou estrutura e expor esse campo de forma segura ao mundo externo.

- Isso é feito usando uma combinação de métodos especiais chamados get e set.

- Propriedades são uma parte fundamental do conceito de encapsulamento em programação orientada a objetos.

# Propriedades

- Como Propriedades Funcionam:
  - Getter (get): O método get é usado para ler o valor de uma propriedade. Quando você tenta acessar o valor da propriedade, o código dentro do get é executado para retornar o valor.
  - Setter (set): O método set é usado para definir o valor de uma propriedade. Quando você atribui um valor a uma propriedade, o código dentro do set é executado para atualizar o valor. O valor atribuído pode ser acessado usando a palavra-chave value dentro do set.

# Propriedades

- Vantagens das Propriedades:
  - Encapsulamento: Você pode controlar como um valor é definido ou retornado (por exemplo, adicionando validação).
  - Simplicidade: As propriedades permitem que campos privados sejam acessados como se fossem públicos, mas com controle adicional.
  - Interoperabilidade: Propriedades são importantes para tecnologias como data binding em frameworks de interface gráfica.

# Propriedade básica

- Uma propriedade básica em C# inclui métodos de acesso get e set que controlam como um campo privado é acessado e modificado.

```
public class Produto
{
    private string _nome;

    //Propriedade básica
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }
}
```

# Propriedade com validação

- Você pode adicionar validação no método set para garantir que os dados sejam sempre válidos, como no exemplo de uma propriedade que não permite nomes vazios ou nulos:

```
public class Produto
{
    private string _nome;

    //Propriedade com validação
    public string Nome
    {
        get { return _nome; }
        set
        {
            if (string.IsNullOrEmpty(value))
            {
                throw new ArgumentException("O nome do produto não pode ser nulo");
            }
            else
            {
                _nome = value;
            }

        }
    }
}
```

# Propridades automáticas

- Se você não precisa de lógica adicional nos métodos get ou set, pode usar propriedades automáticas para uma sintaxe mais simplificada:

```
public class Produto
{
    //Propriedade automática
    public string Nome { get; set; }
}
```

# Propriedades Somente Leitura e Escrita

- Você também pode criar propriedades que são somente de leitura ou somente de escrita usando apenas um dos acessadores get ou set:

```
public class Produto
{
    //Propriedades somente leitura ou somente escrita
    private string _nome;

    private double _preco;

    //propriedade é somente leitura pois não tenho o set
    public string Nome
    {
        get { return _nome; }
        //set { _nome = value; }
    }

    //propriedade é somente escrita pois eu não tenho o get
    public double Preco
    {
        set { _preco = value; }
    }
}
```

# Propriedades Somente Leitura e Escrita

```
public class Produto
{
    //Propriedades somente leitura ou somente escrita

    //Nome será somente leitura
    public string Nome { get; private set; }

    //Preco será somente leitura
    public double Preco { private get; set; }
}
```

# Propriedade com Lógica de get

- Às vezes, você pode querer incluir alguma lógica no método get, como calcular um valor cada vez que ele é acessado:

```
public class Produto
{
    private string _nome;

    private double _preco;
    private double _imposto;

    public string Nome { get => _nome; set => _nome = value; }
    public double Preco { get => _preco; set => _preco = value; }
    public double Imposto { get => _imposto; set => _imposto = value; }

    //Propriedades com lógida de get
    public double PrecoTotal
    {
        get { return Preco + Imposto; }
    }
}
```

# Criando uma propriedade completa

```
public class Pessoa
{
    private string _nome;
    private int _idade;
    private bool _maiorIdade;

    public int Idade
    {
        get
        {
            return _idade;
        }
        set
        {
            if(value < 0)
            {
                throw new ArgumentException("Idade não pode ser negativa");
            }
            _idade = value;
            if (_idade >= 18)
                _maiorIdade = true;
            else
                _maiorIdade = false;
        }
    }
```

# Encapsulamento

- O encapsulamento desempenha um papel fundamental na programação orientada a objetos (POO) em C# por várias razões importantes:

- Segurança e Integridade dos Dados:

  - O encapsulamento permite que os dados de um objeto sejam protegidos de acesso não autorizado ou modificações indevidas. Ao definir campos como privados e expor o acesso a esses campos por meio de propriedades, você pode controlar como os dados são lidos e gravados. Isso ajuda a garantir que os dados permaneçam em um estado válido e consistente.

- Controle e Validação:

  - As propriedades, que encapsulam campos, oferecem a oportunidade de aplicar lógica personalizada ao acessar ou definir valores. Isso permite a validação de dados, como a rejeição de valores inválidos ou a aplicação de regras de negócios específicas antes de alterar o estado do objeto.

# Encapsulamento

- Abstração e Ocultação de Detalhes de Implementação:
  - O encapsulamento permite que você separe a interface pública de uma classe dos detalhes de implementação interna. Isso significa que você pode alterar a forma como a classe funciona internamente sem afetar o código que a utiliza, desde que a interface pública permaneça a mesma. Isso torna o código mais flexível e adaptável a mudanças futuras.
- Facilidade de Manutenção:
  - Quando o acesso aos campos internos de uma classe é controlado por meio de propriedades, você pode fazer alterações na implementação interna sem impactar o restante do código que usa a classe. Isso simplifica a manutenção e a evolução do código ao longo do tempo.

# Encapsulamento

- Reutilização de Código:
  - O encapsulamento promove a reutilização de código, uma vez que a interface pública de uma classe pode ser usada em diferentes partes do programa sem a necessidade de duplicar código. Isso reduz erros e torna o desenvolvimento mais eficiente.
- Leitura e Documentação Claras:
  - Ao acessar os campos internos de uma classe por meio de propriedades, o código fica mais legível, pois os desenvolvedores podem entender facilmente a intenção por trás das operações. Além disso, as propriedades podem ser documentadas para explicar como elas devem ser usadas.

# Encapsulamento

- Polimorfismo:
  - O encapsulamento é fundamental para a aplicação de princípios de POO, como o polimorfismo. Através de classes base e herança, você pode manter uma interface consistente e permitir que diferentes subclasses tenham diferentes implementações internas sem quebrar o contrato com a interface pública.
- Em resumo, o encapsulamento desempenha um papel crucial na POO em C# porque fornece controle sobre o acesso a dados, promove a segurança e a integridade dos dados, facilita a manutenção do código, permite a reutilização de código e apoia os princípios da orientação a objetos, como o polimorfismo e a abstração.
- É uma prática fundamental para escrever código confiável, seguro e flexível.

# Exercício 1

- Crie uma classe Carro com três propriedades: Marca, Modelo e Ano. Use o encapsulamento adequado para garantir que:
  - a) A Marca e o Modelo possam ser lidos e escritos externamente.
  - b) O Ano só possa ser definido no construtor e acessado (mas não modificado) depois disso.

# Exercício 2

- Crie uma classe ContaBancaria com os seguintes atributos: NumeroConta, Saldo e Titular. Criar propriedades para acesso (get e set) aos dados.
  - Analise as necessidades de encapsulamento.
  - Crie os métodos construtor, public bool Depositar(double valor), public bool Sacar(double valor) e public void ExibirSaldo()

# Exercício 3

- 3) Crie uma classe Pessoa com as propriedades Nome e Idade. Adicione validação nas propriedades para que:
  - a) Nome não aceite valores nulos ou strings vazias.
  - b) Idade não aceite valores menores que 0 ou maiores que 120.

# Exercício 4

- Crie uma classe Retangulo com propriedades para Largura e Altura, e uma propriedade calculada Area. A área deve ser calculada como Largura * Altura.

# Exercício 5

- Defina uma classe Estudante que contém as propriedades Nome e uma propriedade privada notaFinal.
    - A notaFinal deve ser acessível apenas através de um método GetNotaFinal(), que deve retornar "Aprovado" se a nota for maior ou igual a 70 e "Reprovado" se for menor