Fundamentos de C#

Encapsulamento e Organização do Código

Softblue
cursos online

Tópicos Abordados

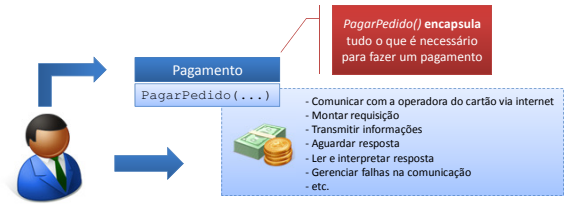
- Encapsulamento
- Modificadores de visibilidade
- Accessors e mutators
- Properties
  - Read-only e write-only properties
  - Auto-implemented properties
  - Static properties
  - Visibilidade
- Object initializers
- Namespaces
  - A diretiva using
  - Namespaces aliases
- Partial classes
- Extension methods

Encapsulamento

- É um dos pilares da orientação a objetos

Encapsulamento

- O princípio do encapsulamento consiste em esconder detalhes de implementação de quem chama o objeto



Encapsulamento

- No encapsulamento, também é importante não expor o estado do objeto de forma direta
  - O estado do objeto é uma informação interna ao objeto

```
class Livro
{
    public string titulo;
    public int numPaginas;
    public double preco;
}
```

Fere o princípio do encapsulamento

A comunicação entre objetos é feita através de **métodos** ou **properties**

Encapsulamento: Por quê? Quando?

- Por que usar o encapsulamento?
  - Define responsabilidades aos objetos
    - Aumenta a coesão
    - Agrupa funcionalidades comuns
  - Esconde a implementação
    - A forma como o objeto manipula os fields e implementa os métodos não interfere em quem usa este objeto
  - Facilita a manutenção do código
    - Mudar fields ou implementação de métodos não "quebra" o código existente
  - Facilita a extensão do código
    - Novos métodos e/ou fields podem ser criados no objeto sem prejuízo à aplicação
- Quando usar o encapsulamento?
  - Sempre

Modificadores de Visibilidade

- Controlam a visibilidade de elementos
  - Tipos de dados
    - Classes
    - Estruturas
    - Enums
  - Membros
    - Fields
    - Properties
    - Métodos
    - Construtores

Modificadores de Visibilidade

Modificador	Significado
private	Visível apenas à classe/estrutura que declarou o elemento.
public	Sem restrição de acesso.
protected	Visível à classe que declarou e também às suas subclasses.
internal	Visível apenas dentro do próprio assembly.
protected internal	Visível à classe que declarou, subclasses e dentro do próprio assembly.

Visibilidade Padrão

- Quando um modificador de visibilidade não é fornecido explicitamente, é assumido um valor padrão

```
class Livro
{
    string titulo;

    Livro()
    {
    }

    void Emprestar()
    {
    }
}
```

internal

private

private

private

Accessors e Mutators

- O princípio do encapsulamento diz que os fields devem ser privados
  - Como acessá-los ou manipulá-los?
- **Accessors** e **mutators** são uma opção
  - Métodos **getters** e **setters**
- Expõem os fields através de métodos

Accessors e Mutators

```
class Livro
{
    private int numPaginas;
    private double preco;
    public int GetNumPaginas()
    {
        return numPaginas;
    }
    public void SetNumPaginas(int numPaginas)
    {
        this.numPaginas = numPaginas;
    }
    public double GetPreco()
    {
        return preco;
    }
    public void SetPreco(double preco)
    {
        this.preco = preco;
    }
}
```

Fields privados

Accessor (getter)

Mutator (setter)

Accessors e Mutators

```
Livro l = new Livro();
l.SetNumPaginas(250);
```

```
int p = l.GetNumPaginas();
```

O acesso aos fields é feito apenas através de métodos

```
public void SetPreco(double preco)
{
    if (preco > 0)
    {
        this.preco = preco;
    }
}
```

Verifica se o parâmetro é consistente

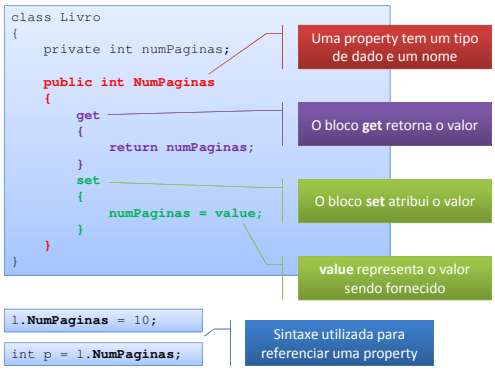
```
l.SetPreco(-10);
```

O valor não é atribuído ao field

Properties

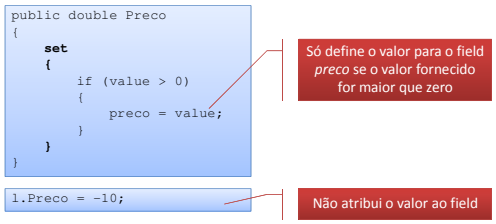
- Recurso presente no C# para encapsular os fields de um objeto
- São a forma recomendada em C# para o encapsulamento de dados
- Têm os mesmos recursos que accessors e mutators
- O acesso é feito como se fosse o acesso direto a um field

Declaração de Properties



Declaração de Properties

- É possível implementar uma lógica de negócio em uma property



Read-Only Properties

- Podem ter o valor lido, mas não alterado

```
public int NumPaginas
{
    get
    {
        return numPaginas;
    }
}
```

Apenas o bloco **get** é especificado

```
int p = l.NumPaginas;
```

```
l.NumPaginas = 250;
```

Erro de compilação

Write-Only Properties

- Podem ter o valor alterado, mas não lido

```
public int NumPaginas
{
    set
    {
        numPaginas = value;
    }
}
```

Apenas o bloco **set** é especificado

```
l.NumPaginas = 250;
```

```
int p = l.NumPaginas;
```

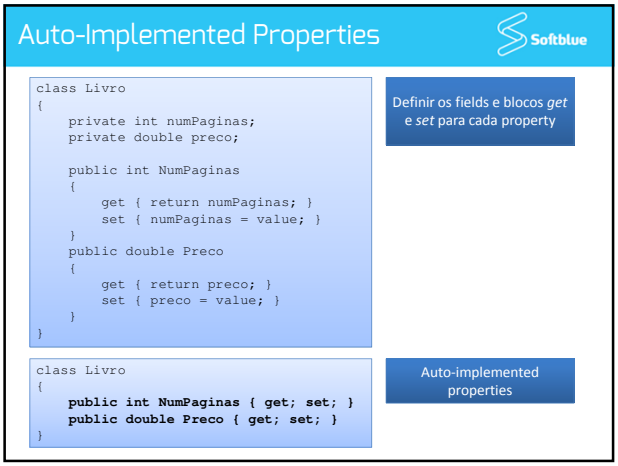
Erro de compilação

Properties que permitem somente escrita não são muito comuns

Auto-Implemented Properties

- Permitem simplificar a criação de properties em alguns casos
  - A property deve suportar leitura e escrita
  - A property não tem qualquer lógica de negócio, a não ser atribuir ou retornar algum valor

Auto-Implemented Properties



```
class Livro
{
    private int numPaginas;
    private double preco;

    public int NumPaginas
    {
        get { return numPaginas; }
        set { numPaginas = value; }
    }
    public double Preco
    {
        get { return preco; }
        set { preco = value; }
    }
}
```

Definir os fields e blocos *get* e *set* para cada property

```
class Livro
{
    public int NumPaginas { get; set; }
    public double Preco { get; set; }
}
```

Auto-implemented properties

Static Properties

- É possível declarar properties como **static**
  - Pertencem à classe, e não aos objetos

```
private static int maxPaginas;

public static int MaxPaginas
{
    get { return maxPaginas; }
    set { maxPaginas = value; }
}
```

Property definida como *static*

```
public static int MaxPaginas { get; set; }
```

Auto-implemented property

Visibilidade em Properties

- Os blocos *get* e *set* de uma property assumem a visibilidade definida na property

```
public int NumPaginas
{
    get { return numPaginas; }
    set { numPaginas = value; }
}
```

get e *set* são **public**

- Os blocos *get* e *set* de uma property podem ter visibilidades diferentes

```
public int NumPaginas
{
    get { return numPaginas; }
    private set { numPaginas = value; }
}
```

get é **public** e *set* é **private**

Visibilidade em Properties

- Um dos blocos deve assumir a visibilidade definida para a property

```
public int NumPaginas
{
    private get { return numPaginas; }
    private set { numPaginas = value; }
}
```

Erro de compilação

Object Initializers

- Permite a criação de um objeto e a definição de valores de fields públicos e properties usando apenas um comando

```
class Livro
{
    public string Titulo { get; set; }
    public int NumPaginas { get; set; }
    public double Preco { get; set; }
}
```

```
Livro l = new Livro();
l.Titulo = "C# na Softblue";
l.NumPaginas = 250;
l.Preco = 80.0;
```

Criação e inicialização do objeto

```
Livro l = new Livro {
    Titulo = "C# na Softblue", NumPaginas = 250, Preco = 80.0 };
```

Object initializer

Object Initializers e os Construtores

- O uso do object initializer também chama o construtor da classe

```
Livro l = new Livro { Titulo = "C# na Softblue" };
Livro l = new Livro() { Titulo = "C# na Softblue" };
```

Invoca o construtor padrão

- Outros construtores podem ser usados

```
Livro l = new Livro(10) { Titulo = "C# na Softblue" };
```

Invoca o construtor que recebe um parâmetro *int*

Namespaces

- Permitem organizar o código de forma lógica

```
namespace Interface
{
    class TelaInicial
    {
        //...
    }
}
```

```
namespace Logica
{
    class Login
    {
        //...
    }
}
```

- O namespace não tem relação com o nome do arquivo *.cs* onde os elementos estão sendo declarados

Namespaces Aninhados

- É possível definir um namespace dentro de outro namespace

```
namespace Softblue
{
    namespace Interface
    {
        class TelaInicial
        {
            //...
        }
    }
}
```

Dois níveis de namespaces

Recomenda-se criar pelo menos dois níveis

```
namespace Softblue.Interface
{
    class TelaInicial
    {
        //...
    }
}
```

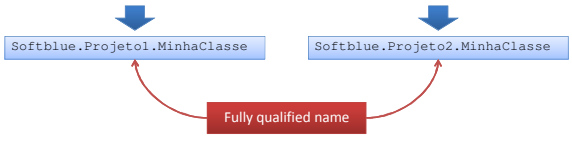
Separação dos níveis através do "."

Namespaces e a Ambiguidade

- O uso de namespaces evita ambiguidade em nomes de elementos

```
namespace Softblue.Projeto1
{
    class MinhaClasse
    {
        //...
    }
}
```

```
namespace Softblue.Projeto2
{
    class MinhaClasse
    {
        //...
    }
}
```



A Diretiva *using*

- Um elemento deve ser referenciado pelo seu fully qualified name

```
Softblue.Projeto1.MinhaClasse c = new Softblue.Projeto1.MinhaClasse();
```

- A diretiva **using** permite importar um namespace

```
using Softblue.Projeto1;
```

- O fully qualified name não é mais necessário

```
MinhaClasse c = new MinhaClasse();
```

Namespace *System*

- O namespace **System** contém classes muito usadas no C#

```
System.Console.WriteLine("Texto");
```

Classe **Console** do namespace **System**

- É muito comum importar este namespace nos arquivos *.cs* do projeto

```
using System;

Console.WriteLine("Texto");
```

Namespaces Aliases

- A diretiva *using* também permite dar um apelido (alias) para um namespace existente

```
using P1 = Softblue.Projeto1;
```

Softblue.Projeto1 pode ser referenciado como **P1**

```
P1::MinhaClasse c = new P1::MinhaClasse();
```

Usa-se "::" em aliases

Mais Detalhes Sobre Namespaces

- Namespaces são lógicos
  - Não estão associados à arquivos de código-fonte ou assemblies
- Além de classes, interfaces, estruturas e enums também podem ser definidos dentro de namespaces
- Um elemento sempre tem um namespace associado
  - Se você não definir um, ele fará parte de um namespace global, sem nome
- Procure sempre definir um namespace
  - Recomenda-se pelo menos dois níveis
    - *<organização>.<projeto>*

Partial Classes

- Uma classe pode ter o seu código-fonte dividido em vários arquivos .cs
- O modificador **partial** é utilizado

Livro-fields.cs

```
namespace Softblue
{
    partial class Livro
    {
        string isbn;
        string titulo;
        string autor;
        int numPaginas;
    }
}
```

Livro-methods.cs

```
namespace Softblue
{
    partial class Livro
    {
        public void Emprestar()
        {
        }

        public void Devolver()
        {
        }
    }
}
```

Partial Classes

- Do ponto de vista de uso da classe, não muda nada
  - É como se todo o código estivesse definido em um arquivo só
- Partial classes devem ser definidas sob o mesmo namespace

Extension Methods

- Permite adicionar métodos a uma classe sem modificar o código-fonte desta classe

```
class Livro
{
    public int NumPaginas { get; set; }
}
```

```
static class LivroExtension
{
    public static void ImprimirPaginas(this Livro l)
    {
        Console.WriteLine(l.NumPaginas);
    }
}
```

Extension Method

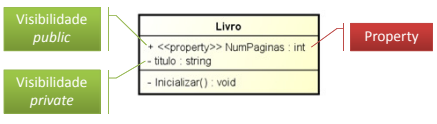
```
Livro l = new Livro();
l.ImprimirPaginas();
```

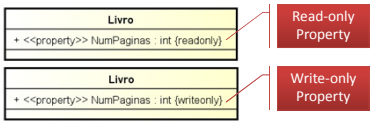
A chamada é feita como se o método pertencesse ao objeto

Extension Methods

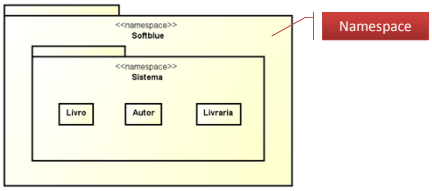
- Um extension method é apenas uma facilidade para o programador
- Quando o código é compilado, é gerada uma chamada estática
- Não prejudica o encapsulamento
  - O extension method não tem acesso aos membros privados do objeto

UML





UML


<<namespace>>
Softblue
<<namespace>>
Sistema
Livro
Autor
Livraria
Namespace


Softblue