Desvendando os Generics no C#: Atuação para Código Reutilizável e Eficiente

Nagib Sabbag Filho

Leaders.Tec.Br, 1(6), ISSN: 2966-263X, 2024.

e-mail: profnagib.filho@fiap.com.br

DOI: https://doi.org/10.5281/zenodo.13328256

PermaLink: https://leaders.tec.br/artigo/desvendando-os-generics-no-c-atuacao-para-codigo-reutilizavel-e-eficiente

Received: 08 Aug 2024 / Accepted: 10 Aug 2024 / Published online: 12 Aug 2024

Abstract:

Generics em C# permitem criar código flexível e reutilizável para diferentes tipos de dados, oferecendo benefícios como reutilização de código, segurança em tempo de compilação e melhorias de desempenho, exemplificados pela criação de listas genéricas, métodos de comparação e uso com restrições de tipo.

Key words:

generics, c#, código reutilizável, tipos genéricos, eficiência, tipos parametrizados, classe genérica, restrições de tipo, generics em .net.

Introdução aos Generics

Generics são uma poderosa funcionalidade do C# que permite aos desenvolvedores criar classes, métodos, interfaces e delegados com um alto grau de flexibilidade e reutilização. Com os Generics, é possível escrever código que pode ser utilizado com diferentes tipos de dados sem a necessidade de duplicação, garantindo uma maior eficiência e manutenção do código.

Benefícios dos Generics

Os Generics oferecem diversos benefícios, incluindo:

• Reutilização de Código: Permitem criar estruturas de dados que funcionam com qualquer tipo específico, evitando a repetição de código.

• Segurança em Tempo de Compilação: Os erros são detectados em tempo de compilação, evitando problemas em tempo de execução.

• Desempenho: Evitam o boxing e unboxing de tipos de valor, o que pode melhorar o desempenho.

Exemplos Práticos de Utilização dos Generics

Vamos explorar alguns exemplos que demonstram a aplicação dos Generics em cenários complexos e recentes.

Exemplo 1: Criação de uma Lista Genérica

```
public class GenericList < T >
{
```

```
        private T[] elements;
        private int count = 0;

        public GenericList(int capacity)
        {
            elements = new T[capacity];
        }

        public void Add(T item)
        {
            if (count < elements.Length)
            {
                elements[count] = item;
                count++;
            }
            else
            {
                throw new InvalidOperationException("List is full");
            }
        }

        public T GetElement(int index)
        {
            if (index >= 0 && index < count)
            {
                return elements[index];
            }
            else
            {
                throw new IndexOutOfRangeException("Index is out of range");
            }
        }
    }
```

Este exemplo demonstra a criação de uma lista genérica que pode armazenar elementos de qualquer tipo especificado no momento da instância.

Exemplo 2: Implementação de um Método Genérico para Comparação

```
    public class GenericComparer
    {
        public bool AreEqual < T > (T value1, T value2)
        {
            return value1.Equals(value2);
        }
    }
```

Este método genérico AreEqual permite comparar dois valores de qualquer tipo, desde que o tipo suporte a operação de igualdade.

Aplicações Avançadas dos Generics

Além dos exemplos básicos, os Generics podem ser aplicados em cenários mais avançados, como a criação de algoritmos genéricos e manipulação de dados complexos.

Exemplo 3: Uso de Generics com Restrições de Tipo

```
    public class DataProcessor < T > where T : IData
    {
        public void ProcessData(T data)
        {
            data.Validate();
            data.Save();
        }
    }

    public interface IData
    {
        void Validate();
        void Save();
    }

    public class CustomerData : IData
    {
        public void Validate()
        {
            // Validação específica para CustomerData
        }

        public void Save()
        {
            // Salvamento específico para CustomerData
        }
    }
```

Neste exemplo, a classe DataProcessor é restrita a tipos que implementam a interface IData, garantindo que os métodos Validate e Save estejam disponíveis.

Exemplo 4: Classe Genérica com Múltiplas Restrições de Tipo

```
    public class Repository < T, TKey >
        where T : class
        where TKey : struct
    {
        private readonly Dictionary < TKey, T > _dataStore = new Dictionary < TKey, T >
();

        public void Add(TKey key, T item)
        {
            _dataStore[key] = item;
        }

        public T Get(TKey key)
        {
            if (_dataStore.TryGetValue(key, out T item))
            {
                return item;
            }
            throw new KeyNotFoundException("Key not found");
```

```
    }
}
```

Neste exemplo, a classe Repository é genérica com duas restrições de tipo: T deve ser uma classe e TKey deve ser um tipo de valor. Isso garante que a chave seja um tipo de valor (por exemplo, int) e o item seja uma referência de classe.

Exemplo 5: Classe Genérica com Restrição de Tipo para Classe Abstrata

```
public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Perimeter { get; }
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.PI * Radius * Radius;
    public override double Perimeter => 2 * Math.PI * Radius;
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public override double Area => Width * Height;
    public override double Perimeter => 2 * (Width + Height);
}

public class ShapePrinter < T >  where T : Shape
{
    public void PrintShapeDetails(T shape)
    {
        Console.WriteLine($"Area: {shape.Area}");
        Console.WriteLine($"Perimeter: {shape.Perimeter}");
    }
}
```

Neste exemplo, a classe ShapePrinter é um genérico que é restrito a tipos que herdam da classe abstrata Shape. A classe ShapePrinter possui um método PrintShapeDetails que escreve as propriedades Area e Perimeter da forma

fornecida.

Conclusão

Os Generics são uma característica fundamental da linguagem C# que proporcionam flexibilidade e segurança no desenvolvimento de software. Ao permitir a criação de código mais genérico e reutilizável, eles ajudam a manter a eficiência e a integridade do sistema. A adoção de Generics pode levar a um código mais limpo, seguro e com melhor desempenho, contribuindo para a criação de aplicações robustas e escaláveis.

Referências

Para aprofundar-se ainda mais sobre Generics no C#, consulte as seguintes fontes oficiais:

• MICROSOFT. Microsoft Docs - Generics in C#. Disponível em: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/. Acesso em: 29 jul. 2024.

• MICROSOFT. Microsoft Docs - .NET Generics. Disponível em: https://learn.microsoft.com/en-us/dotnet/standard/generics/. Acesso em: 29 jul. 2024.

Nagib é Professor Universitário e Tech Manager.
Possui uma trajetória de conquistas em certificações técnicas e ágeis,
incluindo MCSD, MCSA e PSM1.
PG em Gestão de TI pelo SENAC e MBA em Tecnologia de Software pela USP,
Nagib cursou programas de extensão do MIT e Universidade de Chicago.
Outras conquistas incluem a autoria de um artigo sobre chatbots, revisado por pares e apresentado na Universidade de Barcelona.