Unveiling Generics in C#: Working for Reusable and Efficient Code

Nagib Sabbag Filho
FIAP (Faculty of Informatics and Administration Paulista) Avenida Paulista, 1106 - 7º andar - Bela Vista, São Paulo, Brazil.
e-mail: profnagib.filho@fiap.com.br
PermaLink: https://leaders.tec.br/article/890dd5
ago 12 2024

Abstract:
Generics in C# allow for the creation of flexible and reusable code for different data types, offering benefits such as code reuse, compile-time safety, and performance improvements, exemplified by the creation of generic lists, comparison methods, and use with type constraints.

Key words:
generics, c#, reusable code, generic types, efficiency, parameterized types, generic class, type constraints, generics in .net.

Introduction to Generics

Generics are a powerful feature of C# that allows developers to create classes, methods, interfaces, and delegates with a high degree of flexibility and reusability. With Generics, it is possible to write code that can be used with different data types without the need for duplication, ensuring greater efficiency and maintainability of the code.

Benefits of Generics

Generics offer several benefits, including:

Code Reusability: They allow the creation of data structures that work with any specific type, avoiding code repetition.
Compile-Time Safety: Errors are detected at compile time, preventing runtime issues.
Performance: They avoid boxing and unboxing of value types, which can improve performance.

Practical Examples of Using Generics

Let's explore some examples that demonstrate the application of Generics in complex and recent scenarios.

Example 1: Creating a Generic List

```
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");
            }
        }
    }
```

This example demonstrates the creation of a generic list that can store elements of any type specified at the time of instantiation.

Example 2: Implementing a Generic Method for Comparison

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

This generic method AreEqual allows comparing two values of any type, as long as the type supports the equality operation.

Advanced Applications of Generics

In addition to basic examples, Generics can be applied in more advanced scenarios, such as the creation of generic algorithms and manipulation of complex data.

Example 3: Using Generics with Type Constraints

```
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()
        {
            // Specific validation for CustomerData
        }

        public void Save()
        {
            // Specific saving for CustomerData
        }
    }
```

In this example, the DataProcessor class is restricted to types that implement the IData interface, ensuring that the Validate and Save methods are available.

Example 4: Generic Class with Multiple Type Constraints

```
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");
        }
    }
```

In this example, the Repository class is generic with two type constraints: T must be a class and TKey must be a value type. This ensures that the key is a value type (e.g., int) and the item is a class reference.

Example 5: Generic Class with Type Constraint for Abstract Class

```
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}");
        }
    }
```

In this example, the ShapePrinter class is a generic that is restricted to types that inherit from the abstract class Shape. The ShapePrinter class has a method PrintShapeDetails that writes the Area and Perimeter properties of the provided shape.

Conclusion

Generics are a fundamental feature of the C# language that provide flexibility and safety in software development. By

allowing the creation of more generic and reusable code, they help maintain the efficiency and integrity of the system. The adoption of Generics can lead to cleaner, safer, and better-performing code, contributing to the creation of robust and scalable applications.

References

For further in-depth information about Generics in C#, consult the following official sources:

MICROSOFT. Microsoft Docs - Generics in C#. Available at: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/. Accessed on: July 29, 2024.
MICROSOFT. Microsoft Docs - .NET Generics. Available at: https://learn.microsoft.com/en-us/dotnet/standard/generics/. Accessed on: July 29, 2024.

Nagib is a University Professor and Tech Manager. He has a track record of achievements in technical and agile certifications, including MCSD, MCSA, and PSM1. He holds a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP. Nagib has taken extension programs at MIT and the University of Chicago. Other achievements include authorship of a peer-reviewed article on chatbots, presented at the University of Barcelona.