Pattern Matching

Ziggy Rafiq

C#Corner

Pattern Matching in C#

Ziggy Rafiq

Author – Ziggy Rafiq
Publisher – C# Corner
Editorial Team – Deepak Tewatia, Baibhav Kumar
Publishing Team – Praveen Kumar
Promotional & Media – Rohit Tomar

Background and Expertise

Over his 19 years of experience, Ziggy Rafiq has demonstrated exceptional skills in System Architecture. With over 19 years of experience, he is a highly accomplished Full-Stack Designer and Developer. In 2004, he was appointed Technical Lead Developer, highlighting his leadership and expertise.

Several prestigious awards have recognized Ziggy Rafiq's exceptional talents. After he developed an impenetrable login system in 2002, he received the Shell Award for his groundbreaking work. In 2008, he was honored as one of Microsoft's Top 10 Developers in the West Midlands at the Microsoft Hero Event.

Ziggy Rafiq has been recognized by C# Corner as an MVP, VIP, and Member of the Month (July), as well as an active speaker and Chapter Lead at the UK Developer Community.

Educational background

As a student at college, Ziggy Rafiq earned an American Associate Degree in Interactive Multimedia Communication. Continuing his education at the University of Wolverhampton from 1999 to 2003, he earned a BA Hons degree in Interactive Multimedia Communication 2:1. He gained a comprehensive understanding of Design, Development, Testing, Deployment, and Project Management along the way, making him an effective professional.

Acknowledgement

This book is dedicated to author Ziggy Rafiq's late mother, Mrs. Zubeda Begum, whose unwavering love, support, and encouragement have been a constant source of inspiration. From January 1st, 1950, to December 1st, 2022, her presence in Ziggy Rafiq's life shaped Ziggy Rafiq's journey, and this book is a tribute to her memory.

Special Thanks

Ziggy Rafiq extends his heartfelt gratitude to the following individuals and communities who have supported Ziggy Rafiq throughout the journey of creating this book.

Ziggy Rafiq Family and Friends

Ziggy would like to extend his deepest gratitude to his wife, children, and father, Mohammed Rafiq, for their support, patience, and understanding throughout the writing process.

Ziggy Rafiq Advisers

Thanks to Ziggy's mentors and advisers for their guidance and valuable insights, enriching this book.

The Software Engineer Community

Thank you to the vibrant online community of software engineers for sharing knowledge and providing inspiration.

Ziggy Rafiq Late Mother Mrs. Zubeda Begum

Whose legacy of love and encouragement continues to inspire Ziggy Rafiq every day.

Mahesh Chand from C# Corner

Your support and encouragement have been instrumental in making this book a reality.
Ziggy Rafiq, author of this book sincerely thankful to each one of you.

— Ziggy Rafiq

Table of Contents:

1

Introduction

Overview

Using pattern matching in C#, developers can streamline conditional logic by identifying and acting upon patterns within data structures, which is a pivotal advancement in programming. It emphasizes the importance of pattern matching in enhancing code clarity and maintainability, which is explored in this introductory chapter.

Importance and Benefits of Pattern Matching

In addition to improved readability, reduced complexity, and enhanced flexibility, pattern matching offers significant advantages in software development. In addition to facilitating more efficient and reliable coding practices, pattern matching enables precise data manipulation based on structural patterns.

Outline of Different Types of Pattern Matching Covered in the Book

A detailed overview of the various types of pattern-matching techniques explored in the book can be found in this chapter, including:

- Constant Patterns
- Type Patterns
- Property Patterns
- Tuple Patterns
- Recursive Patterns
- Positional Patterns

Practical examples and detailed explanations illustrate each type's implementation and utility in real-world programming.
A comprehensive introduction to pattern matching in C# is presented in these sections, setting the stage for further exploration in subsequent chapters. This chapter provides a comprehensive foundation for mastering pattern matching, whether you're new to it or want to improve your proficiency.

Overview of Pattern Matching in C#

Using pattern matching in C#, developers can efficiently handle different types of data structures and scenarios, allowing them to write more concise and expressive code. Pattern matching was introduced in C# 7.0 and significantly enhanced in later versions, allowing developers to match values against patterns and execute corresponding code. It incorporates sophisticated patterns, such as type patterns, property patterns, and recursive patterns, in addition to simple equality checks.

C#'s pattern matching features are explored in this book and comprehensive guidance is provided on how to take advantage of them effectively. Each chapter discusses different aspects of pattern matching, from syntax and usage to advanced techniques and real-world examples, demonstrating its versatility across a wide range of scenarios, from control flow to data manipulation. This book provides you with the knowledge and practical skills to fully utilize C# pattern matching, regardless of whether you are new to the feature or seeking a deeper understanding.

Importance and Benefits of Pattern Matching

In modern software development, pattern matching in C# is crucial because it enhances code clarity, maintainability, and functionality. Pattern matching promotes clean and readable code by allowing developers to express conditional logic based on the structure and shape of data. As a result of this clarity, errors are less likely to occur and debugging is easier, leading to easier maintenance and extension of codebases.

By streamlining common programming tasks, pattern matching boosts developer productivity. Using it, you can handle a wide variety of data patterns, from simple types to complex hierarchical structures without having to resort to nested conditional statements or excessive type checking. By optimizing how data is processed and manipulated, this efficiency not only reduces boilerplate code but also improves performance.

In addition to its technical advantages, pattern matching promotes best practices for software design. By supporting declarative and functional programming, it aligns with modern software engineering paradigms that emphasize clarity, modularity, and scalability. A pattern-matching approach can help developers create systems that are flexible and resilient enough to adapt gracefully to changing requirements and evolving data models.

The importance of pattern matching lies in its ability to elevate code quality, improve developer productivity, and foster robust software design. Developers can create efficient, maintainable, and scalable applications by mastering pattern matching in C#.

Outline of Different Types of Pattern Matching Covered in the Book

switch statement

The switch statement allows you to match specific constant values. This is particularly useful when you need to execute different code blocks based on the exact value of a variable.

```
 namespace PatternMatchingLibrary;
public class DayOfWeekChecker
{
    public void CheckDay(DayOfWeek dayOfWeek)
    {
        switch (dayOfWeek)
        {
            case DayOfWeek.Monday:
                Console.WriteLine("Today is Monday");
                break;
            case DayOfWeek.Friday:
                Console.WriteLine("Today is Friday");
                break;
            default:
                Console.WriteLine("It's neither Monday nor Friday");
                break;
        }
    }
}
```

Explanation

With the switch statement, the current value of the variable dayOfWeek is evaluated. The cases within the switch specify constant values against which dayOfWeek is compared (specifically, DayOfWeek.Monday and DayOfWeek.Friday in this example). The code block associated with this case is executed if the value of dayOfWeek matches dayOfWeek.Monday, which in this case outputs "Today is Monday." Additionally, if dayOfWeek matches DayOfWeek.Friday, the code block that prints "Today is Friday" executes. When dayOfWeek does not match either of these specific values, the default case is triggered, displaying "It's neither Monday nor Friday".

Benefits

One of the advantages of switch statements with specific constant values is that they allow you to manage multiple conditional branches using predefined values such as DayOfWeek.Monday or DayOfWeek.Friday in a clear and concise manner. As a result of this clarity, code readability is improved because each case explicitly states its expected values and actions. As a result of

using switch statements, logical errors are less likely to occur when complex nested conditions are encountered with multiple if statements. Using the switch structure, every case handles a distinct condition, allowing precise control over program flow.

Use Cases

Using this pattern, you can execute distinct actions or produce specific outputs based on predefined constant values in particular scenarios. It is ideal for implementing logic that relies on well-known and fixed values, such as days of the week (e.g., DayOfWeek.Monday, DayOfWeek.Friday), status codes (e.g., HTTP status codes), or enumeration constants (e.g., enumeration constants representing different states or types). Using a switch statement with constant values allows you to clearly define and manage each possible scenario within your application, ensuring accuracy and efficiency.

Type Patterns

C# provides type patterns that can match and work with objects based on their runtime types. This is particularly useful when you need to handle different types of objects in a polymorphic way.

```
using PatternMatchingLibrary.Components;
using PatternMatchingLibrary.Components.Base;
using Rectangle = PatternMatchingLibrary.Components.Rectangle;

namespace PatternMatchingLibrary;

public class ShapeInfoPrinter
{
    public void PrintShapeInfo(Shape shape)
    {
        if (shape is Circle circle)
        {
            Console.WriteLine($"Circle with radius {circle.Radius}");
        }
        else if (shape is Rectangle rectangle)
        {
            Console.WriteLine($"Rectangle with width {rectangle.Width}
and height {rectangle.Height}");
        }
        else
        {
            Console.WriteLine("Unknown shape type");
        }
    }
}

namespace PatternMatchingLibrary.Components.Base;
public abstract class Shape
{
    public abstract double Area();
}
using PatternMatchingLibrary.Components.Base;
```

```
namespace PatternMatchingLibrary.Components;
public class Circle : Shape
{
    public double Radius { get; }

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

    public override double Area()
    {
        return Math.PI * Radius * Radius;
    }
}
```

Code Example can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/Chapter01/PatternMatchingLibrary/Components/Circle.cs

```
using PatternMatchingLibrary.Components.Base;

namespace PatternMatchingLibrary.Components;

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

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

    public override double Area()
    {
        return Width * Height;
    }
}
```

Explanation

The is keyword is used in the provided example to determine whether an object is circular or rectangular. When the shape is identified as a circle, the condition shape is Circle circle evaluates to true, allowing the variable circle to be assigned the casted value of shape. As a result, the program outputs details specific to circles, such as their radius, within the corresponding if block (Console.WriteLine($"Circle with radius [circle.Radius]);). Alternatively, if shape does not match the Circle type, then the else if statement verifies that shape conforms to

the Rectangle type. Upon a positive verification, the casted value is assigned to the variable rectangle. In this else if block, the program prints specific details related to a rectangle, including its width and height (Console.WriteLine($"Rectangle with width [rectangle.Width] and height [rectangle.Height]);).

Benefits

A polymorphic approach to object management in C# allows developers to execute different behaviours depending on the object. It enhances flexibility and adaptability in code design by simplifying the implementation of varied operations for different object types.

The use of type patterns improves the readability of code by indicating the intended actions of objects based on their runtime types. With this approach, manual type checks and casting are reduced, resulting in a codebase that is easier to maintain and more comprehensible.
The type of pattern simplifies the consolidation of logic that manages diverse object types into cohesive and efficient blocks, reducing code duplication. A developer can streamline development efforts and ensure consistent behaviour across different parts of an application by eliminating redundant code structures. Software quality and development productivity are improved by this practice, which promotes code reuse and scalability.

Use Cases

The type patterns are especially useful for calculating circles, rectangles, triangles, and more when dealing with various shapes. By identifying and applying appropriate behaviours based on the runtime type of the shape object, they enable streamlined operations specific to each shape type, such as calculating area or rendering. With this approach, geometric entities are handled efficiently and tailored within applications, improving their user experience and functionality.
In applications handling diverse data types like different file formats or database entities, type patterns provide a structured approach to uniform handling and processing based on the data encountered at runtime. Applications can ensure consistent and reliable data management across multiple sources and formats by dynamically discerning and processing data types.
Multiple plugins may conform to a common interface but offer distinct functionalities based on their types, so type patterns play a crucial role in plugin architectures. By allowing applications to load and execute plugins dynamically based on runtime types, modular software can be designed and extended. Plugin-based systems can be made more flexible, scalable, and maintainable by leveraging type patterns to seamlessly integrate new features and enhancements without modifying the core logic.

Property Patterns

When objects are matched according to their properties in C#, they provide a flexible approach to conditional logic that takes specific object attributes into account.

```
using PatternMatchingLibrary.Components.Base;
using PatternMatchingLibrary.Components;

namespace PatternMatchingLibrary;

public static class ShapeMatcher
{
    public static void MatchShape(Shape shape)
    {
        switch (shape)
        {
            case Circle { Radius: > 10 }:
```

```
                Console.WriteLine("Large circle with radius greater
than 10");
                break;
            case Rectangle { Width: var w, Height: var h } when w == h:
                Console.WriteLine("Square with side length equal to
width and height");
                break;
            default:
                Console.WriteLine("Shape with unknown properties");
                break;
        }
    }
}
```

Explanation

According to the provided example, a switch statement evaluates the shape of an object and applies specific patterns based on its properties. Circle [Radius: > 10]: pattern matches circles with a radius of greater than 10, displaying a message. Rectangle [Width: var w, Height: var h] when w == h: pattern identifies squares with equal width and height, printing a message corresponding to those squares. For shapes that don't fit these patterns, the default: case takes care of instances with unknown properties, ensuring that different shapes can be handled comprehensively.

Benefits

As a result of property patterns, objects can be precisely matched based on their attributes, enabling custom handling of different situations. Property patterns reduce the need for complex nested conditions and enhance code clarity by specifying conditions directly within switch cases based on object properties. Furthermore, this approach allows for flexible conditional logic integration, making it easier to maintain and more expressive.

Use Cases

Various scenarios can benefit from property patterns, including data filtering and processing tasks. They enable graphical applications to filter objects based on specific attribute values, such as numerical thresholds or matching dimensions. In configurations and settings management, property patterns facilitate dynamic adjustments based on identifiable object properties, ensuring configurations are tailored to specific conditions. Property patterns also simplify the validation process against custom rules or business logic tied to object properties, facilitating custom validation and rule checking. By minimizing the reliance on complex if-else structures, this approach promotes more efficient and expressive code, which enhances code clarity and maintainability.

Tuple Patterns

The tuple pattern provides a concise and expressive way to handle multiple patterns of structured data in C#.

```
namespace PatternMatchingLibrary.Models;
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public Point(int x, int y)
```

```
    {
        X = x;
        Y = y;
    }
}
var point = new Point(3, 4);
switch (point)
{
    case Point p when p.X == 0 && p.Y == 0:
        Console.WriteLine("Origin");
        break;
    case Point p when p.X != 0 && p.Y == 0:
        Console.WriteLine($"Point on x-axis at {p.X}");
        break;
    case Point p when p.X == 0 && p.Y != 0:
        Console.WriteLine($"Point on y-axis at {p.Y}");
        break;
    case Point p when p.X == p.Y:
        Console.WriteLine($"Point on diagonal at ({p.X}, {p.Y})");
        break;
    default:
        Console.WriteLine($"Arbitrary point ({point.X}, {point.Y})");
        break;
}
```

Explanation

As shown in the example, a tuple is defined as (3, 4), which the switch statement evaluates. Patterns with zero elements identify the origin and print "Origin" (Console.WriteLine("Origin"); in the case (0, 0). In the case (var x, 0), the pattern matches tuples with zero elements, indicating a point on the x-axis, and outputs the x-coordinate (Console.WriteLine($"Point on x-axis at [x]"); In the case (0, var y), the pattern matches tuples with zero elements, indicating a point on the y-axis, and outputs the y-coordinate (Console.Writ The case var (x, y) when x == y: pattern matches tuples where both elements are equal, representing a point on the diagonal, and prints the coordinates (Console.WriteLine($"Point on diagonal at ([x], [y])"));). In the absence of any of these patterns, the default case executes, printing a message indicating an arbitrary point (Console.WriteLine($"Arbitrary point ([point. Item1], [point.Item2]));).

Benefits

In addition to providing an efficient method of handling and processing structured data represented as tuples, tuple patterns can also be used to match and process specific patterns of data easily. Developers can implement concise and readable code that addresses various scenarios based on the values within tuples by supporting pattern matching directly on tuples. The tuple pattern also enhances code clarity and maintainability by encapsulating multiple cases in a single switch statement, reducing the complexity of conditional logic. As a result, the development process is simplified and structured data in programming tasks is handled robustly.

Use Cases

With a wide range of applications, tuple patterns are particularly useful in coordinate systems, categorizing points based on their position relative to axes, or specific patterns, such as origin, axes, or diagonals. By filtering or processing data based on specific combinations of values

stored in tuples, tuple patterns simplify targeted data processing in data-centric applications. Furthermore, tuple patterns are instrumental in algorithmic implementations where distinct behaviour or operations are required based on the configuration of tuple data, thus facilitating the design of efficient and structured algorithms. In addition to improving code clarity and efficiency, this versatility ensures robust handling of structured data and complex situations.

Recursive Patterns

By using recursive patterns in C#, hierarchical data structures can be matched recursively, providing a powerful mechanism for handling nested or hierarchical data structures

namespace PatternMatchingLibrary;

```
public static class NumberSummarizer
{    public static int SumNumbers(object obj)
    {
        switch (obj)
        {
            case int num:
                return num;
            case IEnumerable<int> numbers:
                return numbers.Sum();
            case IEnumerable<object> objects:
                return objects.OfType<int>().Sum(o => SumNumbers(o));
            default:
                return 0;
        }
    }
}
```

Explanation

SumNumbers is designed to calculate the sum based on the type of object passed as obj in the provided example. By evaluating the type of object, the switch statement determines the appropriate calculation method. When obj is an int, the case int num: pattern matches, returning the integer value directly. When obj is of type IEnumerable<int>, such as a collection of integers, the switch matches the case IEnumerable<int> numbers: pattern, utilizing LINQ's Sum() method to calculate the sum of these integers. For IEnumerable<object> objects, which represent a collection of objects, the method recursively applies SumNumbers to each object within the collection and sums the results. Those objects whose patterns do not match the default case are returned with 0. With the SumNumbers method, you can handle multiple kinds and collections of data in an efficient, straightforward manner.

Benefits

Recursive patterns offer several benefits: They simplify the handling of nested or hierarchical data structures by providing a straightforward approach to applying patterns recursively within nested collections or structures. By doing so, developers are able to manage complex data hierarchies without explicit iteration or traversal, resulting in more concise and maintainable code. By defining intricate matching logic that adapts dynamically to the level of nesting or hierarchy of data, recursive patterns also support flexible data processing scenarios. Due to this flexibility, pattern matching can be applied to a wide range of programming tasks, ensuring efficient and structured data handling across diverse fields.

Use Cases

A recursive pattern can be used in a variety of scenarios: It excels in managing nested data structures such as trees or deeply nested collections, in which elements themselves may contain further nested elements. Recursive patterns are particularly effective for navigating and processing hierarchical data representations efficiently because of this capability. By simplifying operations such as summing values, calculating averages, or extracting specific elements from nested collections or structures, recursive patterns facilitate data transformation and aggregation. Additionally, recursive patterns enable the implementation of complex algorithms requiring recursive traversal or processing of hierarchical data, improving the clarity and efficiency of algorithmic designs across a wide range of programming tasks. Regardless of the application domain, this versatility ensures robust handling and processing of structured data.

Positional Patterns

By using positional patterns in C#, developers can match objects based on the structure of their classes or structs, making pattern matching more concise and expressive.

```
using PatternMatchingLibrary.Models;

namespace PatternMatchingLibrary;

public static class PersonMatcher
{
    public static void MatchPerson(Person person)
    {
        switch (person)
        {
            case Person("Lisa", "Smith"):
                Console.WriteLine("Welcome, Lisa Smith!");
                break;
            case Person(var firstName, var lastName):
                Console.WriteLine($"Hello, {firstName} {lastName}");
                break;
            default:
                Console.WriteLine("Unknown person");
                break;
        }
    }
}

namespace PatternMatchingLibrary.Models;

public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
```

```
        LastName = lastName;
    }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}
```

Explanation

"John" is assigned to the firstName and "Doe" is assigned to the lastName in the provided example. The switch statement evaluates the structure of the person object. The case Person ("John", "Doe"): pattern matches instances where the constructor parameters of the person object are "John" and "Doe". The program outputs a personalized welcome message within this case block (Console.WriteLine("Welcome, John Doe!") ;). Any instance of Person with the firstName and lastName parameter is captured by the case Person (var firstName, var lastName): pattern. Based on the values captured within this case block, the program prints a generic greeting (Console.WriteLine($"Hello, [firstName] [lastName]") ;). If person does not match either of these specific patterns, the default: case executes, printing a message indicating an unknown person (Console.WriteLine("Unknown person"); The structured approach allows for tailored handling of Person objects based on specific constructor parameter values, resulting in clear and concise conditional logic.

Benefits

In addition to structural matching, positional patterns offer direct matching based on the constructor parameters of classes or structures. By encapsulating logic within switch cases, this approach simplifies pattern matching, improves code readability and reduces the complexity of conditional checks. Furthermore, positional patterns improve code maintainability by expressing the intent of matching based on object structure, which contributes to clarity and reduces potential errors in code maintenance.

Use Cases

Various software development use cases are well suited to positional patterns. In constructor parameter matching scenarios, where objects' states are defined by specific constructor parameters, they enable precise initialization of objects with tailored attributes or properties. By validating and processing objects according to predefined constructor parameter values, positional patterns play an essential role in data validation applications, ensuring robust data integrity and consistency checks. Moreover, positional patterns streamline the handling of structured data in the context of configuration or settings objects, enabling straightforward implementation of application behaviours and configurations driven by predefined values or structures.
To help you master pattern matching in C#, we provide practical examples, best practices, and advanced techniques to help you master these types of pattern matching. It explores different scenarios in which pattern matching can be applied effectively, allowing you to write cleaner, more concise, and more maintainable code in each chapter. Whether you're new to pattern matching or looking to deepen your understanding, this book will guide you through exploring this powerful feature in C#.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

2

Introduction to Pattern Matching

Overview

In modern C# programming, pattern matching has revolutionized the way developers deal with conditional logic and data manipulation. In this chapter, we cover the essentials of pattern matching, its syntax, and its evolution within C#.

We will explore pattern matching's origins in functional programming languages and its evolution into mainstream languages like C# in this chapter. By matching and extracting data based on its structure, type, or shape, pattern matching offers developers a more concise and readable alternative to conditional statements.

Key Topics Covered

What is Pattern Matching?

- C# pattern matching definition and importance.
- The benefits of pattern matching for code clarity and maintainability.

Evolution and Adoption of Pattern Matching

- Historically, pattern matching has been adopted across programming languages in a variety of contexts.
- C# versions have been enhanced with pattern-matching features.

Basic Concepts and Syntax in C# for Pattern Matching

- An introduction to type patterns, property patterns, tuple patterns, and switch statements with patterns.
- Pattern matching simplifies complex data-handling scenarios using practical examples.

Why This Chapter Matters

Developing effective C# code requires understanding pattern matching. It enables developers to develop code that is more expressive and robust, thus enhancing productivity and scalability. By the end of this chapter, readers will have a solid grasp of pattern matching fundamentals and be well-equipped to apply these concepts effectively throughout the book.

Who Should Read This Chapter

For beginners and experienced developers alike, this chapter provides a thorough introduction to pattern matching in C#, as well to deepen their knowledge. In subsequent chapters, advanced topics will be explored by focusing on pattern matching. This chapter provides a comprehensive overview of pattern matching's capabilities.
Throughout the chapter, practical examples and insights ensure readers not only understand the syntax of pattern matching but also appreciate its strategic importance in modern software development practices. This chapter provides a solid foundation for effectively leveraging pattern matching in C#, whether you're just getting started or seeking to improve your programming proficiency.

What is pattern matching?

As a fundamental programming technique, pattern matching allows developers to identify patterns within data structures and take actions based on them. Pattern matching in C# provides a way to match and extract data from objects based on their shape, structure, or type, thereby making your code more concise and readable.
It involves comparing data structure with a pattern and executing code based on the match. A hierarchical data structure can be used to match specific values, types, properties, tuples, or even more complex patterns. Developers can replace nested conditional statements with more expressive and maintainable code with pattern matching.
It was introduced in C# to simplify common programming tasks, improve code clarity, and make applications more robust. Since its introduction in C# 7.0, it has evolved significantly, providing

developers with a versatile toolkit to handle a wide variety of scenarios, from simple control flow to advanced data manipulation.
Throughout this chapter, we will explore the foundational concepts of pattern matching, its syntax, and basic usage scenarios. The purpose of this presentation is to demonstrate how pattern matching improves readability and flexibility in your code, making it easier to manage and adapt to changing requirements. You'll find in this chapter an overview of pattern matching in C# programming that will serve you well whether you're new or looking to deepen your understanding.

Evolution and Adoption of Pattern Matching in Programming Languages

In modern programming paradigms, pattern matching has evolved from its origins in functional programming languages into a cornerstone feature. By matching data's structure against predefined patterns, pattern matching provided a concise and expressive way to handle data rooted in languages such as ML and Haskell.
With time, pattern matching gained traction and made its way into mainstream languages such as C#, F#, Scala, and Swift. As pattern matching evolved in each language, its capabilities expanded beyond traditional functional programming domains to fit its own syntax and semantics.
It was part of a broader effort to enhance language capabilities and improve developer productivity that pattern matching was formally introduced in C# version 7.0. As C# has evolved, its pattern matching features have been refined and expanded across subsequent versions, incorporating new patterns and optimizations to make it more powerful and versatile.
As a result of pattern matching's value in simplifying complex conditional logic and improving code clarity, these languages have adopted it. It facilitates cleaner and more maintainable codebases by enabling developers to express intricate data-handling scenarios concisely. With its evolution, it underscores the importance of declarative, expressive, and scalable programming as a fundamental tool for modern software development.
Our goal in this chapter is to explain how pattern matching has been integrated into C# over the years, how it has evolved, and how it can be applied to real-world programming. We will demonstrate how pattern matching enhances the flexibility and robustness of C# applications, enabling developers to write more efficient and readable code through examples and explanations.

Basic Concepts and Syntax in C# for Pattern Matching

A pattern match in C# helps developers write cleaner, more expressive code by introducing a versatile approach to conditional statements. Here are the fundamental concepts and syntax elements you need to know:

Type Patterns

With type patterns in C#, code can be executed conditionally based on an object's runtime type. In the following example, the is keyword is used to check if the object is a Circle object. If the condition shape is Circle circle evaluates to true, it means shape is indeed a Circle object, and the casted value is assigned to the variable circle. As a result, the if block can access properties that pertain to the Circle class directly. By using Console.WriteLine($"Circle with radius [circle.Radius]"), the program prints out the circle's radius.

```
if (shape is Circle circle)
{
    Console.WriteLine($"Circle with radius {circle.Radius}");
}
```

In situations where different actions or behaviors need to be performed based on the specific type of an object, type patterns improve code clarity by eliminating explicit type casting and allowing more concise and readable conditional logic. It is particularly useful in polymorphic scenarios, where different types of objects may share a common interface or base class, but require distinct handling depending on their characteristics.

Property Patterns

The switch statement evaluates the shape object to determine which case matches based on its properties. Property patterns in C# allow developers to match objects based on specific properties and their values.

```
switch (shape)
{
    case Circle { Radius: > 10 }:
        Console.WriteLine("Large circle with radius greater than 10");
        break;
    case Rectangle { Width: var w, Height: var h } when w == h:
        Console.WriteLine("Square with side length equal to width and
height");
        break;
}
```

A case case of Circle [Radius: > 10]: matches instances where the Radius property of a shape object is greater than 10. When this condition is met, the corresponding case block executes, which includes the statement Console.WriteLine("Large circle with radius greater than 10"));. The message indicates that the shape is a large circle with a radius greater than 10. By using this approach, developers are able to handle different cases or conditions in a structured and readable manner using C# programming.

When the width and height of a shape object are equal, the second case, Rectangle [Width: var w, Height: var h] when w h, matches the shape object of type Rectangle. As soon as this condition is satisfied, the corresponding case block executes, containing the statement Console.WriteLine("Square with side length equal to width and height");. This outputs a message indicating that the shape is a square with side length equal to width and height. In C#, property patterns allow developers to define conditions based on object properties, enhancing the clarity and structure of code handling different object types and states.

With property patterns, conditional logic can be streamlined by comparing and checking object properties within switch cases directly. By encapsulating complex property-based conditions into concise and expressive blocks, this approach improves code readability and maintainability, making it easier to handle different scenarios based on specific object attributes. The technique is particularly useful in scenarios where objects have well-defined properties that determine their behavior or outcome.

Tuple Patterns

Developers can match tuples of specific values efficiently using tuple patterns in C#. In this example, a tuple point is defined as (10, 14).

[need to start from here code example]

```
var pointTuple = (10, 14);
string location = pointTuple switch
{
    (0, 0) => "Origin",
```

```
    (var x, 0) => $"Point on x-axis at {x}",
    (0, var y) => $"Point on y-axis at {y}",
    _ => $"Arbitrary point ({pointTuple.Item1}, {pointTuple.Item2})"
};
Console.WriteLine(location);
```

When both elements of a tuple equal 0, this pattern indicates that the tuple represents the origin. The program prints "Origin" inside this case block using Console.WriteLine("Origin");.case (var x, 0): This pattern matches tuples in which the second element equals 0, indicating an x-axis position. It is possible to print a message indicating the x-coordinate of the point on the x-axis by capturing the value of the first element (Console.WriteLine($"Point on x-axis at [x]");).case (0, var y): It matches tuples whose first element is 0, indicating a position on the y-axis. The value of the second element y is captured and used to print a message indicating the y-coordinate of the point on the y-axis (Console.WriteLine($"Point on y-axis at {y}");).default: If the tuple point does not match any of the specific patterns above, default is used. The message indicates an arbitrary point using the values of the tuple elements (Console.WriteLine($"Arbitrary point ([point.Item1], [point.Item2]));.

In scenarios where data is represented in pairs or tuples and specific patterns of values must be identified and processed, tuple patterns are particularly useful. It enhances the clarity and maintainability of code dealing with structured data by offering a concise and readable way to handle different cases based on the structure and values of tuples.

Switch Statements with Patterns

In the provided switch statement example utilizing patterns. Case Circle c when c.Radius > 10: This pattern matches instances where c.Radius is greater than 10 and the shape is Circle. In this case block (Console.WriteLine("Large circle");), the program outputs "Large circle," indicating a circle with a radius greater than 10. In this pattern, Rectangle [Width: var w, Height: var h] matches instances where both the width (w) and the height (h) of the shape are equal. The program prints "Square" within this case block (Console.WriteLine("Square");), forming a square with equal width and height.default: If none of the specific patterns above match the shape, the default case is executed. The console prints "Unknown shape" (Console.WriteLine("Unknown shape");), indicating that the shape does not meet the preceding patterns' conditions.

```
var shape2 = new Circle(radius: 45);
string shapeType = shape2 switch
{
    Circle c when c.Radius > 10 => "Large circle",
    _ => "Unknown shape"
};
Console.WriteLine(shapeType);
```

This approach demonstrates how switch statements in C# can leverage patterns directly to handle different types and properties of objects succinctly and clearly. Patterns enable the code to be more expressive and maintain readability while efficiently handling different cases based on the input shape structure and conditions.

Recursive Patterns

Using recursive patterns in C# allows for recursive application of patterns to elements in hierarchical data structures. A switch statement evaluates obj to determine its type and then performs specific actions according to that type. A sumNumbers method accepts an object parameter obj, which can represent a variety of data types. When obj is of type int, this pattern matches. In this case, it directly returns the integer value num. case IEnumerable<int> numbers:

This pattern matches when obj is an IEnumerable<int>, indicating a collection of integers. It calculates the sum of these integers using LINQ's Sum() method. case IEnumerable<object> objects: This pattern matches when obj is an IEnumerable<object>, representing a collection of objects. This method recursively applies SumNumbers to each object in the collection (objects.Sum(o => SumNumbers(o))) and sums them. The default case executes if obj does not match any of the specific patterns above, returning 0. As a result, if obj is an unrecognized type or an empty collection, the method gracefully handles it by returning a default value.

```
namespace PatternMatchingLibrary;
public static class Numbers
{
    public static int SumNumbers(object obj)
    {
        return obj switch
        {
            int num => num,
            IEnumerable<int> numbers => numbers.Sum(),
            IEnumerable<object> objects => objects.Sum(o =>
SumNumbers(o)),
            _ => 0
        };
    }
}
object objectOne = 44;
object objectTwo = new List<int> { 17, 52, 23 };
object objectThree = new List<object> { 4, new List<int> { 5, 6 }, 7 };

Console.WriteLine(Numbers.SumNumbers(objectOne));
Console.WriteLine(Numbers.SumNumbers(objectTwo));
Console.WriteLine(Numbers.SumNumbers(objectThree));
```

In general, recursive patterns in C# are an efficient and concise way of handling nested or hierarchical data structures. Using switch statements, developers can define recursive matching logic directly within switch statements, reducing the need for explicit iteration or traversal of complex data hierarchies. In data structures exhibiting nested relationships or varying levels of depth, this approach improves code clarity, maintainability, and flexibility.

C# pattern matching allows you to handle diverse data patterns with clarity and efficiency, which is demonstrated in these examples. To help you leverage pattern matching in your C# applications effectively, we'll dive deeper into each pattern type, examine advanced techniques, and provide practical insights. This chapter serves as a comprehensive introduction to mastering C# pattern matching, whether you're a new user or seeking to refine your skills.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

3

Constant Pattern Matching

Overview

C# supports constant pattern matching as a foundational aspect of pattern matching, which allows developers to match specific values within their code with constant values. This chapter explores the intricacies of constant patterns, providing detailed explanations, syntax examples, and practical applications to demonstrate their importance and effectiveness in modern software development.

Understanding Constant Patterns

Developers can apply constant patterns to switch statements and pattern-matching expressions in C# to match values against specific constants. This type of pattern matching is straightforward and useful when comparing exact values.

Syntax and Usage Examples

In C#, constant patterns are defined by using the case keyword followed by the constant value.

```
namespace PatternMatchingLibrary;
public class DayOfWeekChecker
{

    public string GetDayOfWeek(DayOfWeek dayOfWeek)
    {
        string message = dayOfWeek switch
        {
            DayOfWeek.Monday => "It's Monday!",
            DayOfWeek.Friday => "It's Friday!",
            _ => "It's another day."
        };

        return message;
    }
}

var dayChecker = new DayOfWeekChecker();
Console.WriteLine(dayChecker.GetDayOfWeek(DayOfWeek.Monday));
Console.WriteLine(dayChecker.GetDayOfWeek(DayOfWeek.Friday));
Console.WriteLine(dayChecker.GetDayOfWeek(DayOfWeek.Wednesday));
```

In this code example, the switch statement evaluates the dayOfWeek parameter. The case DayOfWeek.Monday: and case DayOfWeek.Friday: lines demonstrate constant pattern matching against specific days of the week. The default: case handles days that are not explicitly matched.

Benefits and Practical Applications

- **Readability** Defining meaningful constant names makes your code more readable and maintainable.
- **Maintainability:** It reduces errors by ensuring that only expected values are processed.
- **Switch Statement Clarity:** Using constant patterns in switch statements improves clarity and reduces nesting, making code easier to understand and maintain.

Best Practices for Using Constant Patterns Effectively

- **Use Descriptive Constants:** Define meaningful constant names to improve code readability and maintainability.
- **Handle All Cases:** To handle unexpected values gracefully, include a default case or _ pattern.
- **Combine with Other Patterns:** To create more complex matching scenarios, constant patterns can be combined with type patterns or property patterns.

- **Avoid Overuse:** Use other pattern-matching techniques for more dynamic scenarios and avoid overusing constant patterns.

By enabling developers to match specific constant values efficiently, C#'s constant pattern matching feature enhances code clarity and reliability. In this chapter, we have explored constant patterns in depth, including their syntax, usage examples, benefits, practical applications, and best practices. With knowledge of constant patterns, developers can write cleaner, more maintainable code and lay the groundwork for understanding more advanced pattern-matching techniques.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

4

Type Pattern Maching

Overview

C#'s type pattern matching allows developers to match and extract values based on their type, making it an effective tool for handling polymorphic situations. The intricacies of type patterns are discussed in this chapter, along with their practical applications and considerations for leveraging them effectively in software development.

Syntax and Usage Examples

When dealing with polymorphic hierarchies, such as inheritance and interfaces, type patterns make it easy to match objects based on their type. Here's how type patterns look in C#:

```
using PatternMatchingLibrary.Components;
using PatternMatchingLibrary.Components.Base;

namespace PatternMatchingLibrary;

public class ShapeInfoPrinter
{

    public string DescribeShape(Shape shape)
    {
        string description = shape switch
        {
            Circle circle => $"Circle with radius {circle.Radius}",
            Rectangle rectangle => $"Rectangle with width
{rectangle.Width} and height {rectangle.Height}",
            null => "Shape is null",
            _ => "Unknown shape"
        };

        return description;
    }
}

 ShapeInfoPrinter shapeInfoPrinter = new ShapeInfoPrinter();

Circle circleTypePatternMatching = new Circle(45);
Rectangle rectangleTypePatternMatching = new Rectangle(13,54);
Shape? nullShapeTypePatternMatching = null;


Console.WriteLine(shapeInfoPrinter.DescribeShape(circleTypePatternMatch
ing));
Console.WriteLine(shapeInfoPrinter.DescribeShape(rectangleTypePatternMa
tching));
Console.WriteLine(shapeInfoPrinter.DescribeShape(nullShapeTypePatternMa
tching));

Console.ReadLine();
```

In the code example above, we have the switch statement evaluates the shape parameter. The case Circle circle: and case Rectangle rectangle: lines demonstrate type pattern matching against specific subtypes of the Shape base class. Null: handle the case where the shape is null. The default: case handles any other shape that isn't explicitly matched.

Pattern Matching with Types and Subtypes

They serve as a concise means of distinguishing between different implementations of the same base class or interface. They can be applied to base types, derived types, or interfaces:

```
namespace PatternMatchingLibrary.Components.Base;
public abstract  class Vehicle
{
    public string Manufacturer { get; set; } = string.Empty;
    public string Model { get; set; } = string.Empty;
    public int Year { get; set; } = 0;
    public abstract string GetDescription();
}

 using PatternMatchingLibrary.Components.Base;
namespace PatternMatchingLibrary.Components;
public class Bicycle : Vehicle
{
    public int NumWheels { get; set; }
    public override string GetDescription()
    {
        return $"This is a Bicycle standard description by Ziggy Rafiq
and Bicycle with {NumWheels}";
    }
}

using PatternMatchingLibrary.Components.Base;
namespace PatternMatchingLibrary.Components;
public class Car : Vehicle
{
    public int NumberOfDoors { get; set; }
    public override string GetDescription()
    {
        return $"This is a car standard description by Ziggy Rafiq and
car with {NumberOfDoors}";
    }
}
using PatternMatchingLibrary.Components.Base;
namespace PatternMatchingLibrary.Components;
public class Truck : Vehicle
{
    public double PayloadCapacity { get; set; }

    public override string GetDescription()
    {
        return "This is a truck standard description by Ziggy Rafiq";
    }
}
```

```
using PatternMatchingLibrary.Components.Base;
using PatternMatchingLibrary.Components;
namespace PatternMatchingLibrary;
public class VehicleProcessor
{
    public string ProcessVehicle(Vehicle vehicle)
    {
        return vehicle switch
        {
            Car car => $"Car with {car.NumberOfDoors} doors",
            Truck truck => $"Truck with payload capacity of
{truck.PayloadCapacity} tons",
            Bicycle bicycle => $"Bicycle with {bicycle.NumWheels}
wheels",
            _ => "Unknown vehicle type"
        };
    }
}
VehicleProcessor processor = new VehicleProcessor();

Car car = new Car
{
    Manufacturer = "Toyota",
    Model = "Camry",
    Year = 2024,
    NumberOfDoors = 4
};

Console.WriteLine(car.GetDescription());
Console.WriteLine(processor.ProcessVehicle(car));

Truck truck = new Truck
{
    Manufacturer = "Ford",
    Model = "F-150",
    Year = 2012,
    PayloadCapacity = 3.5
};

Console.WriteLine(truck.GetDescription());
Console.WriteLine(processor.ProcessVehicle(truck));

Bicycle bicycle = new Bicycle
{
    Manufacturer = "Schwinn",
    Model = "Mountain Bike",
    Year = 2023,
    NumWheels = 2
```

```
};
Console.WriteLine(bicycle.GetDescription());
Console.WriteLine(processor.ProcessVehicle(bicycle));
```

Using Type Patterns with Inheritance and Interfaces

Inheritance hierarchies and interface implementations seamlessly integrate with type patterns, allowing developers to handle polymorphic objects without explicit type checks or casting:

```
namespace PatternMatchingLibrary.Components.Base;

public interface IShape
{
    string ShapeType { get; }
}
using PatternMatchingLibrary.Components.Base;

namespace PatternMatchingLibrary.Components;
public class Circle : Shape, IShape
{
    public double Radius { get; set; }
    public string ShapeType => "Circle";

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

    public override double Area()
    {
        return Math.PI * Radius * Radius;
    }
}
using PatternMatchingLibrary.Components.Base;

namespace PatternMatchingLibrary.Components;

public class Rectangle : Shape, IShape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public string ShapeType => "Rectangle";
    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }
        public override double Area()
    {
        return Width * Height;
```

```
    }
}

using PatternMatchingLibrary.Components.Base;

namespace PatternMatchingLibrary.Components
{
    public class Triangle: Shape, IShape
    {
        public double Base { get; set; }
        public double Height { get; set; }
        public string ShapeType => "Triangle";
        public override double Area()
        {
            return Base * Height;
        }
        public Triangle(double @base, double height)
        {
            Base = @base;
            Height = height;
        }
    }
}

using PatternMatchingLibrary.Components;
using PatternMatchingLibrary.Components.Base;

namespace PatternMatchingLibrary;

public class ShapeInfoPrinter
{

    public string GetShapeDescription(IShape shape)
    {
        return shape switch
        {
            Circle circle when circle.Radius > 10 => $"Large circle
with radius {circle.Radius}",
            Rectangle rectangle when rectangle.Width ==
rectangle.Height => $"Square with side length {rectangle.Width}",
            Triangle triangle => $"Triangle with base {triangle.Base}
and height {triangle.Height}",
            _ => "Unknown shape"
        };
    }

}
IShape[] shapes =
```

```
            [
                new Circle(radius: 5),
                new Rectangle(width: 4, height: 4),
                new Triangle(@base: 3, height: 5)
            ];

foreach (var shapeInfo in shapes)
{
    Console.WriteLine(shapeInfoPrinter.GetShapeDescription(shapeInfo));
}
```

Advanced Scenarios and Considerations

- **Pattern Matching with Null:** To handle situations where the object reference is null, type patterns can check for null values (case null:).
- **Combining Type Patterns with Property Patterns:** Type patterns can be combined with property patterns for more specific matching criteria.
- **Performance Considerations:** Type patterns should be used judiciously in performance-critical scenarios to avoid unnecessary type checks.

By allowing developers to handle polymorphic scenarios effectively, type pattern matching in C# enhances code clarity and flexibility. This chapter explores type patterns comprehensively, covering their syntax, examples of inheritance and interfaces, advanced scenarios, and suggestions for optimal use. By mastering type patterns, developers can write more expressive and maintainable code, setting the stage for exploring more advanced pattern-matching techniques in later chapters.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

5

Property Pattern Matching

Overview

A property pattern matching feature in C# allows developers to match objects according to the values of their properties, extending the capabilities of pattern matching. This chapter provides an in-depth exploration of property patterns, including syntax, practical applications, and scenarios where they excel in enhancing code clarity and flexibility.

Matching Based on Properties of Objects

A property pattern allows precise matching between objects based on the values of their properties. This is particularly useful when differentiating objects that share a base or interface, but have different properties:

```
using PatternMatchingLibrary.Components.Base;
using PatternMatchingLibrary.Components;

namespace PatternMatchingLibrary;
public static class ShapeMatcher
{

 public static string EvaluateShape(Shape shape)
{
    return shape switch
    {
        Circle { Radius: > 10 } => "Large circle",
        Rectangle { Width: var w, Height: var h } when w == h =>
"Square",
        Rectangle { Width: var w, Height: var h } => $"Rectangle with
width {w} and height {h}",
        _ => "Unknown shape"
    };
}

}

Shape[] shapeInformation =
{
    new Circle(radius: 15),
    new Rectangle(width: 10, height: 10),
    new Rectangle(width: 8, height: 6),

};

foreach (var shapeDetials in shapeInformation)
{
    var description = ShapeMatcher.EvaluateShape(shapeDetials);
    Console.WriteLine($"Shape: {shapeDetials.GetType().Name},
Description: {description}");
}
Console.WriteLine("Please Click Any Key To Carry On Seeing the Code
Examples");
Console.ReadLine();
```

As shown in the code example above, the switch statement evaluates the shape parameter. Patterns such as [Radius: > 10] match circles with a radius greater than 10. Patterns such as {Width: var w, Height: var h} match rectangles based on specific width and height conditions.

Patterns with Tuple Elements and Named Fields

It is also possible to apply property patterns to tuples and objects with named fields, allowing for more flexibility in matching structured data:

```
 using PatternMatchingLibrary.Models;

namespace PatternMatchingLibrary;

public static class PersonMatcher
{

    public static string EvaluatePerson((string firstName, string
lastName) person)
    {
        return person switch
        {
            ("Admin", "Doe") => "Welcome, Mr. Smith!",
            (var first, var last) when first.Length + last.Length > 10
=> $"Long name: {first} {last}",
            _ => "Unknown person"
        };
    }
}

 var person1 = ("Mike", "Smith");
var person2 = ("Lisa", "Hills");
var person3 = ("Lerzan", "Hakim");

Console.WriteLine(PersonMatcher.EvaluatePerson(person1));
Console.WriteLine(PersonMatcher.EvaluatePerson(person2));
Console.WriteLine(PersonMatcher.EvaluatePerson(person3));
Console.WriteLine("Please Click Any Key To Carry On Seeing the Code
Examples");
Console.ReadLine();
```

Examples of Using Property Patterns in Real-World Scenarios

Consider a scenario where different types of vehicles exhibit a variety of properties that influence program behavior.

```
using PatternMatchingLibrary.Components.Base;
namespace PatternMatchingLibrary.Components;
public class Car : Vehicle
{
    public int NumberOfDoors { get; set; }
    public string EngineType { get; set; }=string.Empty;
    public override string GetDescription()
    {
```

```
        return $"This is a car standard description by Ziggy Rafiq and
car with {NumberOfDoors}";
    }
}

 using PatternMatchingLibrary.Components.Base;
using PatternMatchingLibrary.Components;
namespace PatternMatchingLibrary;
public class VehicleProcessor
{

    public  string DescribeVehicle(Vehicle vehicle)
    {
        return vehicle switch
        {
            Car { NumberOfDoors: 4, EngineType: "Electric" } =>
"Electric car with 4 doors",
            Truck { PayloadCapacity: > 5000 } => "Heavy-duty truck",
            Bicycle { NumWheels: 2 } => "Bicycle",
            _ => "Unknown vehicle type"
        };
    }
}

 var vehicles = new Vehicle[]
        {
            new Car { NumberOfDoors = 4, EngineType = "Electric" },
            new Truck { PayloadCapacity = 6000 },
            new Bicycle { NumWheels = 2 },
            new Car { NumberOfDoors = 2, EngineType = "Gasoline" }
        };

foreach (var vehicle in vehicles)
{
    Console.WriteLine(processor.DescribeVehicle(vehicle));
}
```

With property pattern matching in C#, objects can be matched based on their property values, improving code expressiveness and maintainability. This chapter has explored property patterns comprehensively, covering their syntax, usage examples with tuples and named fields, and real-world scenarios where they are especially useful. By understanding property patterns, developers can handle complex data structures and write more robust, flexible code in C#, laying the foundation for exploring advanced pattern-matching techniques and optimizations in subsequent chapters.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

6

Switch Expressions

Overview

In this chapter, we explore Compared to traditional switch statements, switch expressions in C# offer improved readability and flexibility as a modernized approach to conditional logic. In this chapter, we examine switch expressions in detail, focusing on their syntax, use as pattern matching tools, and advantages over traditional switch statements.

Comparing Switch Statements and Switch Expressions

As a staple of conditional logic in C#, switch statements have been used for years, but switch expressions offer several advantages:

- **Conciseness:** Switch expressions are more concise and expressive, reducing boilerplate code.
- **Return Type Inference:** Switch expressions infer the return type, thereby eliminating the need for explicit return statements.
- **Pattern Matching:** Switch expressions seamlessly integrate with pattern matching, allowing for more sophisticated conditional checks.

Syntax and Usage of Switch Expressions for Pattern Matching

Using pattern matching, switch expressions execute corresponding expressions based on values. Here's an example:

```
using PatternMatchingLibrary.Components.Base;
using PatternMatchingLibrary.Components;
namespace PatternMatchingLibrary;
public class VehicleProcessor
{
    public string ProcessVehicle(Vehicle vehicle)
    {
        return vehicle switch
        {
            Car car => $"Car with {car.NumberOfDoors} doors",
            Truck truck => $"Truck with payload capacity of
{truck.PayloadCapacity} tons",
            Bicycle bicycle => $"Bicycle with {bicycle.NumWheels}
wheels",
            _ => "Unknown vehicle type"
        };
    }

    public  string DescribeVehicle(Vehicle vehicle)
    {
        return vehicle switch
        {
            Car { NumberOfDoors: 4, EngineType: "Electric" } =>
"Electric car with 4 doors",
            Truck { PayloadCapacity: > 5000 } => "Heavy-duty truck",
            Bicycle { NumWheels: 2 } => "Bicycle",
            _ => "Unknown vehicle type"
        };
    }
}
```

We have the switch expression above evaluating the vehicle parameter. Each case specifies a pattern to match against specific vehicle types or properties. The _ => "Unknown vehicle" handles any cases that aren't explicitly matched.

Switch Expressions vs. Traditional Switch Statements

Compared to traditional switch statements, switch expressions offer several advantages:

- **Improved Readability:** Switch expressions are more readable and concise, making code easier to understand and maintain.
- **Type Safety:** By providing compile-time checks, switch expressions ensure type safety and reduce runtime errors.
- **Pattern Matching Integration:** Integrating pattern matching with switch expressions enables more expressive and flexible conditional logic.

By combining the familiar switch statement syntax with enhanced readability and flexibility, C# switch expressions provide a modernized approach to conditional logic. Throughout this chapter, we have examined switch expressions comprehensively, covering their syntax, their use for pattern matching, and their advantages over traditional switch statements. In C#, mastering switch expressions enables developers to write cleaner, more expressive code as well as utilize advanced pattern-matching techniques. In real-world applications, this knowledge serves as a foundation for optimizing code efficiency and enhancing software design.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

7

Recursive Pattern Matching

Overview

By enabling developers to match nested structures and collections using recursive pattern matching in C#, pattern matching capabilities are extended. We explore the intricacies of recursive patterns in this chapter, covering their syntax, practical applications, and performance optimization considerations.

Matching Nested Structures and Collections

With recursive patterns, we can match data structures based on their elements or properties, such as arrays, lists, or custom collections.

```
using PatternMatchingLibrary.Components.Base;
using PatternMatchingLibrary.Components;

namespace PatternMatchingLibrary;
public static class ShapeMatcher
{

    public static string ProcessShape(Shape shape)
    {
        string description = shape switch
        {
            Circle circle => $"Circle with radius {circle.Radius}",
            Rectangle rectangle => $"Rectangle with width
{rectangle.Width} and height {rectangle.Height}",
            _ => "Unknown shape"
        };
        return description;
    }

    public static string ProcessShapes(IEnumerable<Shape> shapes)
    {
        string description = shapes switch
        {
            Shape[] array when array.Length == 0 => "Empty shape
array",
            Shape[] array => $"Shape array with {array.Length}
elements",
            List<Shape> list when list.Count == 0 => "Empty shape
list",
            List<Shape> list => $"Shape list with {list.Count}
elements",
            _ => "Unknown shapes collection"
        };
        return description;
    }

}

Shape[] shapesArray = new Shape[]
        {
            new Circle(13),
            new Rectangle(14,37)
        };
```

```
foreach (var shapeList in shapesArray)
{
    Console.WriteLine(ShapeMatcher.ProcessShape(shapeList));
}

List<Shape> shapesList = new List<Shape>
        {
            new Circle(19),
            new Rectangle(34,23)
        };

Console.WriteLine(ShapeMatcher.ProcessShapes(shapesList));
Console.WriteLine("Please Click Any Key To Carry On Seeing the Code
Examples");
Console.ReadLine();
```

This code example above uses Recursive patterns like Shape [] [Length: 0] to match an empty array. It matches lists of shapes that have specific numbers of elements. The _ => "Unknown shape" matches any shapes that have not been explicitly matched.

Recursive Patterns with Arrays, Lists, and Custom Collections

This flexibility allows developers to handle complex data structures efficiently using recursive patterns, which can be applied to arrays, lists, and custom collections.

```
 namespace PatternMatchingLibrary
{
    public class DataAnalyzer
    {
        public string AnalyzeData(IEnumerable<object> data)
        {
            string result = data switch
            {
                null => "Data is null",
                object[] array when array.Length == 0 => "Empty array",
                List<int> list when list.Count > 10 => $"List of
integers with more than 10 elements",
                List<string> list => $"List of strings with
{list.Count} elements",
                _ => "Other collection type"
            };
            return result;
        }
    }

}
List<int> intList = new List<int> { 1, 2, 3, 4, 5 };
List<string> stringList = new List<string> { "apple", "banana",
"cherry" };
object[] emptyArray = Array.Empty<object>();
```

```
IEnumerable<object> nullData = null;

var analyzer = new DataAnalyzer();
Console.WriteLine(analyzer.AnalyzeData(intList.Cast<object>()));
Console.WriteLine(analyzer.AnalyzeData(stringList));
Console.WriteLine(analyzer.AnalyzeData(emptyArray));
Console.WriteLine(analyzer.AnalyzeData(nullData));
Console.WriteLine("Please Click Any Key To Carry On Seeing the Code
Examples");
Console.ReadLine();
```

Practical Examples and Performance Considerations

- **Hierarchical Data Processing:** Recursive patterns offer concise and expressive match capabilities when dealing with hierarchical or nested data structures.
- **Performance Considerations:** Despite their power, recursive patterns should be used judiciously in performance-critical scenarios to optimize matching efficiency and reduce overhead.

By matching nested structures and collections based on their elements or properties, recursive pattern matching in C# enhances code expressiveness and flexibility. In this chapter, recursive patterns are explored in detail, including their syntax, practical examples with arrays, lists, and custom collections, and performance considerations. In real-world applications, the ability to master recursive patterns paves the way for more robust and efficient software solutions.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

8

Pattern Combinators

Overview

In this chapter, we explore combining multiple patterns to create complex matching scenarios, pattern combinators in C# extend the capabilities of pattern matching. The purpose of this chapter is to provide insight into pattern combinators, their syntax, practical applications, and guidelines for effective use.

Using AND, OR, and NOT Combinators

Developers can use pattern combinators to combine patterns using logic operators such as and, or, and not, allowing them to define intricate matching criteria.

```
namespace PatternMatchingLibrary;

    public static string EvaluateNumber(int number)
    {
        string result = number switch
        {
            >= 1 and <= 10 => "Number is between 1 and 10",
            < 0 or > 100 => "Number is less than 0 or greater than
100",
            not 42 => "Number is not 42",
            _ => "Number doesn't match any specified condition"
        };
        return result;
    }

}
Console.WriteLine(NumberSummarizer.EvaluateNumber(15));
Console.WriteLine(NumberSummarizer.EvaluateNumber(-1));
Console.WriteLine(NumberSummarizer.EvaluateNumber(42));
Console.WriteLine(NumberSummarizer.EvaluateNumber(15));
Console.WriteLine(NumberSummarizer.EvaluateNumber(200));
Console.WriteLine(NumberSummarizer.EvaluateNumber(100));
Console.ReadLine();
```

This code example displays numbers between 1 and 10. We also have points less than 0 and beyond 100 that match numbers less than 0 or greater than 100 and not 42 that matches numbers not explicitly matched.

Practical Applications and Case Studies

When matching criteria must be precise and flexible, pattern combinators are a valuable tool, such as:

- **Data Validation:** Validating input based on multiple criteria.
- **State Management:** Application state management: Handling state transitions.
- **Error Handling:** Handling errors based on specific conditions.

As an example of how pattern combinators can be used to validate user input, consider the following:

```
 namespace PatternMatchingLibrary;
public static class UserInputValidation
{
    public static string ValidateUserInput(string input)
    {
        string validationMessage = input switch
```

```
        {
            null or "" => "Input cannot be null or empty",
            { Length: > 20 } => "Input length exceeds 20 characters",
            "admin" or "administrator" => "Reserved username",
            _ => "Input is valid"
        };
        return validationMessage;
    }
}

Console.WriteLine(UserInputValidation.ValidateUserInput(null));
Console.WriteLine(UserInputValidation.ValidateUserInput(""));
Console.WriteLine(UserInputValidation.ValidateUserInput("Valid
input"));
Console.WriteLine(UserInputValidation.ValidateUserInput("Too long input
that exceeds 20 characters"));
Console.WriteLine(UserInputValidation.ValidateUserInput("admin"));
Console.WriteLine(UserInputValidation.ValidateUserInput("administrator"
));
Console.WriteLine("Please Click Any Key To Carry On Seeing the Code
Examples");
Console.ReadLine();
```

Guidelines for Effectively Using Pattern Combinators

- **Clarity and Readability:** By succinctly expressing complex matching conditions, combinators enhance code clarity and readability.
- **Avoid Over-Complexity:** It is important to maintain a balance between the creation of expressive patterns and the readability of the code.
- **Test Coverage:** Coverage of complex pattern combinations should be ensured through comprehensive testing.
- **Performance Considerations:** Consider performance implications, especially when combining multiple patterns.

As a result of pattern combinators in C#, code expressiveness and flexibility are enhanced by creating sophisticated matching scenarios. Pattern combinators have been explored comprehensively in this chapter, covering their syntax, practical applications, and guidelines. When developers master pattern combinators, they can create more robust and maintainable software solutions in diverse real-world applications by handling complex conditional logic effectively.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

9

Pattern Matching in LINQ Queries

Overview

By allowing developers to filter and process data based on complex matching conditions, pattern matching in LINQ queries enhances data retrieval and manipulation. LINQ queries are integrated with pattern matching in this chapter, which discusses syntax, practical examples, and performance optimization considerations.

Matching Patterns in Data Retrieval and Manipulation

By leveraging the full power of C#'s pattern matching capabilities, LINQ enables developers to apply conditions directly within query expressions:

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;

var specialNumbers = from num in numbers
                     where num >= 5 && num <= 8
                     select num;

Console.WriteLine("Even Numbers:");
foreach (var num in evenNumbers)
{
    Console.Write(num + " ");
}
Console.WriteLine();


Console.WriteLine("Special Numbers (between 5 and 8 inclusive):");
foreach (var num in specialNumbers)
{
    Console.Write(num + " ");
}
Console.WriteLine();
```

A simple condition is used in the code above to filter even numbers. A pattern combinator is applied to match numbers between 5 and 8 inclusively, described as where num %2 == 0.

LINQ to Objects and LINQ to SQL Pattern Matching Examples

Data querying and manipulation across different data sources can be made consistent with pattern matching, both in LINQ to Objects and LINQ to SQL scenarios:

```
using PatternMatchingLibrary.Models;
namespace PatternMatchingLibrary;

public static class PersonMatcher
{

    public static IEnumerable<Person> GetPersonsStartingWithA()
    {
        var persons = new List<Person>
        {
            new Person("Ziggy","Rafiq"),
            new Person("Andrew","Smith"),
```

```
            new Person("Alice", "Joes")
        };
        return from person in persons where
person.FirstName.StartsWith("A") select person;
    }
}

var names = new List<string> { "Alice", "Bob", "Charlie", "David",
"Eve" };
var shortNames = from name in names
                 where name.Length < 5
                 select name;

Console.WriteLine("Short Names (length < 5):");
foreach (var name in shortNames)
{
    Console.Write(name + " ");
}
Console.WriteLine();

var personList = PersonMatcher.GetPersonsStartingWithA();
Console.WriteLine("The Persons who name start with A are as following
below.");
foreach (var person in personList)
{
    Console.WriteLine($"{person.FirstName}{person.LastName}");
}
Console.WriteLine("Please Click Any Key To Carry On Seeing the Code
Examples");
```

Performance Considerations and Best Practices

- **Query Optimization:** Optimize LINQ queries by understanding how patterns and conditions translate into SQL queries.
- **Indexing:** Optimize query performance by indexing database columns used in pattern-matching conditions.
- **Materialization:** When using LINQ to Objects, be mindful of materialization issues to avoid unnecessary memory consumption.

Through pattern matching, developers can apply complex matching conditions directly to query expressions, extending LINQ's capabilities. LINQ pattern matching has been explored comprehensively in this chapter, covering its syntax, practical examples using LINQ to Objects and LINQ to SQL, and optimization considerations. With mastery of pattern matching in LINQ, developers can efficiently retrieve and manipulate data in various scenarios, enhancing the robustness and efficiency of data-driven applications.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

10

Pattern Matching in Async Methods

Overview

By applying patterns to asynchronous operations, pattern matching in async methods enhances error handling and data processing capabilities. By examining syntax, practical examples, error handling strategies, and considerations for handling advanced patterns and pitfalls, this chapter explores how pattern matching can be integrated into async methods.

Matching Patterns with Async Methods and Tasks

By pattern matching in async methods, developers can handle asynchronous results and exceptions with precision, improving code readability and maintainability:

```
namespace PatternMatchingLibrary.Data;
public class NotFoundException: Exception
{
    public NotFoundException() { }
    public NotFoundException(string message) : base(message) { }
    public NotFoundException(string message, Exception innerException)
: base(message, innerException) { }
}
using System.Net;

namespace PatternMatchingLibrary.Data;

public class Fetcher
{
    private readonly HttpClient _httpClient;

    public Fetcher(HttpClient httpClient)
    {
        _httpClient = httpClient ?? throw new
ArgumentNullException(nameof(httpClient));
    }

    public async Task<string> FetchDataAsync(string url)
    {
        try
        {
            var response = await _httpClient.GetAsync(url);

            return response.StatusCode switch
            {
                HttpStatusCode.OK => await
response.Content.ReadAsStringAsync(),
                HttpStatusCode.NotFound => throw new
NotFoundException(),
                _ => throw new HttpRequestException($"Request failed
with status code {response.StatusCode}")
            };
        }
        catch (HttpRequestException ex)
        {
            return $"HTTP Request Error: {ex.Message}";
        }
        catch (NotFoundException ex)
        {
```

```
            return $"Data Not Found Error: {ex.Message}";
        }
        catch (Exception ex)
        {
            return $"Error occurred: {ex.Message}";
        }
    }
}
string url = "https://ziggyrafiq.com";
using var httpClient = new HttpClient();
var dataFetcher = new PatternMatchingLibrary.Data.Fetcher(httpClient);
try
{
    string result = await dataFetcher.FetchDataAsync(url);
    Console.WriteLine($"Data fetched successfully:\n{result}");
}
catch (Exception ex)
{
    Console.WriteLine($"Error occurred: {ex.Message}");
}
```

We have shown above how the switch expression handles the different HTTP status codes returned by an async operation. The catch block uses pattern matching to handle specific exceptions (HttpRequestException and DataNotFoundException) based on pattern matching conditions. throw new DataNotFoundException() demonstrates throwing a custom exception based on pattern matching conditions.

Error Handling and Pattern Matching in Async Operations

Often, async methods handle exceptions asynchronously and apply pattern matching to differentiate between different error conditions:

```
namespace PatternMatchingLibrary;

public static class Numbers
{
    public static async Task<int> DivideAsync(int numerator, int
denominator)
    {
        try
        {
            return await Task.Run(() =>
            {
                if (denominator == 0)
                    throw new DivideByZeroException();

                return numerator / denominator;
            });
        }
        catch (DivideByZeroException)
```

```
        {
            return -1;
        }
    }
}

try
{
    Console.WriteLine("Example 1:");
    Console.WriteLine($"Result is {await Numbers.DivideAsync(14, 3)}");

    Console.WriteLine("Example 2:");
    Console.WriteLine($"Result is {await Numbers.DivideAsync(15, 2)}");

    Console.WriteLine("Example 3:");
    Console.WriteLine($"Result is {await Numbers.DivideAsync(45,5)}");
}
catch (Exception ex)
{
    Console.WriteLine($"Error occurred: {ex.Message}");
}

Console.WriteLine("Please Click Any Key To Carry On Seeing the Code
Examples");
Console.ReadLine();
```

Advanced Patterns and Pitfalls to Avoid

- **Complex Patterns:** In order to maintain code clarity and performance, use caution when applying complex patterns in async methods.
- **Async/Await Best Practices:** To avoid deadlocks and ensure responsive application behaviour, follow best practices for async/await usage.
- **Exception Handling:** Use pattern matching to handle specific exceptions and improve fault tolerance by implementing robust error-handling strategies.

In async methods, pattern matching provides a powerful way to handle asynchronous results and exceptions with precision and clarity. Async pattern matching has been discussed comprehensively in this chapter, including its syntax, practical examples with tasks and async operations, error-handling strategies, and considerations for advanced patterns and pitfalls. When developers master pattern matching in async methods, they will be able to write resilient and efficient asynchronous code, improving the performance and reliability of real-world asynchronous applications.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

11

Advance Topics in Pattern Matching

Overview

Using C#, we examine advanced pattern-matching techniques. By using tuples and deconstruction, pattern matching allows developers to extract and match structured data efficiently. Using tuple patterns and switch statements with when clauses, for example, can improve code clarity and flexibility. Additionally, C#'s pattern-matching capabilities extend to handling nullable types and option types, enabling robust error handling and null-checking mechanisms. We explore future enhancements to C# pattern matching, emphasising its integration with evolving language features and fostering developer adoption.

Pattern Matching with Tuples and Deconstruction

Using C# pattern matching, you can extract and match structured data based on tuples and deconstructions:

```
namespace PatternMatchingLibrary;

public class TupleProcessor
{
    public string ProcessTuple((int x, int y) point)
    {
        string quadrant = point switch
        {
            (0, 0) => "Origin",
            (var x, var y) when x > 0 && y > 0 => "First quadrant",
            (var x, var y) when x < 0 && y > 0 => "Second quadrant",
            (var x, var y) when x < 0 && y < 0 => "Third quadrant",
            (var x, var y) when x > 0 && y < 0 => "Fourth quadrant",
            (_, _) => "On axis"
        };
        return quadrant;
    }
}
var tupleProcessor = new TupleProcessor();
Console.WriteLine($"Point {point1}:
{tupleProcessor.ProcessTuple(point1)}");
Console.WriteLine($"Point {point2}:
{tupleProcessor.ProcessTuple(point2)}");
Console.WriteLine($"Point {point3}:
{tupleProcessor.ProcessTuple(point3)}");
Console.WriteLine($"Point {point4}:
{tupleProcessor.ProcessTuple(point4)}");
Console.WriteLine($"Point {point5}:
{tupleProcessor.ProcessTuple(point5)}");
Console.WriteLine($"Point {point6}:
{tupleProcessor.ProcessTuple(point6)}");
Console.WriteLine($"Point {point7}:
{tupleProcessor.ProcessTuple(point7)}");
```

To enhance code expressiveness and clarity, use tuple patterns to match specific values or ranges within tuples.

Pattern Matching in Switch Statements with When Clauses

The switch statement in C# supports when clauses for additional conditional checks, allowing for more precise pattern matching:

```
using PatternMatchingLibrary.Models;

namespace PatternMatchingLibrary;
```

```
public static class PersonMatcher
{

    public static string EvaluateGrade(int score)
    {
        string grade = score switch
        {
            >= 90 => "A",
            >= 80 and < 90 => "B",
            >= 70 and < 80 => "C",
            >= 60 and < 70 => "D",
            < 60 => "F"
        };
        return grade;
    }
}
```

Code flexibility and maintainability can be improved by using when clauses to switch cases based on specific criteria.

Pattern Matching with Nullable Types and Option Types

C#'s pattern matching effectively handles nullable types and option types, enhancing null-checking and error handling:

```
public string FormatName(string? firstName, string? lastName)
{
    string fullName = (firstName, lastName) switch
    {
        (null, null) => "Unknown",
        (var fn, null) => fn,
        (null, var ln) => ln,
        (var fn, var ln) => $"{fn} {ln}"
    };
    return fullName;
}
Console.WriteLine($"Score {95}: {PersonMatcher.EvaluateGrade(95)}");
Console.WriteLine($"Score {85}: {PersonMatcher.EvaluateGrade(85)}");
Console.WriteLine($"Score {75}: {PersonMatcher.EvaluateGrade(75)}");
Console.WriteLine($"Score {65}: {PersonMatcher.EvaluateGrade(65)}");
Console.WriteLine($"Score {55}: {PersonMatcher.EvaluateGrade(55)}");

try
{
    Console.WriteLine($"Score {105}:
{PersonMatcher.EvaluateGrade(105)}");
}
catch (ArgumentOutOfRangeException ex)
```

```
{
    Console.WriteLine(ex.Message);
}
```

- **Nullable Types:** Pattern matching is used to handle null values within tuples or deconstructed values safely.
- **Option Types:** Use pattern matching to handle optional values effectively, ensuring robust error handling and null checking.

Exploring the Future of Pattern Matching in C#

- **Pattern Matching Enhancements:** Stay up to date on future C# releases for potential enhancements and new features in pattern matching.
- **Integration with Language Features:** See how pattern matching integrates with other features and improvements in the C# language.
- **Community and Adoption:** Promote pattern matching use among the C# community by sharing insights and best practices.

C#'s advanced pattern-matching topics provide developers with advanced techniques for handling structured data, conditional checks, and nullable types. We have examined pattern matching with tuples, deconstruction, switch statements with when clauses, handling nullable and option types, and how pattern matching will evolve in the future. Advanced pattern-matching techniques enable developers to write concise, robust, and maintainable code, leveraging the full power of pattern matching to address complex programming challenges in modern C# development.

Code Example for this book can be found here on my GitHub Repository:

https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples/

12

Final Thoughts

Overview

From foundational concepts to advanced techniques, we have covered all aspects of pattern matching in C# in this chapter. The following sections introduce the syntax and benefits of pattern matching and discuss the types of patterns matching available, including constants, types, and properties. To improve code conciseness and handle complex data structures effectively, switch expressions and recursive patterns were discussed. In addition to pattern combinators, LINQ integration, and asynchronous method handling, advanced topics highlighted pattern matching's versatility across different programming environments. In the practical tips, it was emphasized how important it is to practice regularly, explore advanced patterns, and stay on top of the latest developments in C#. Developers can significantly improve code readability, flexibility, and maintainability by mastering these concepts, empowering them to tackle diverse programming challenges confidently and efficiently.

Summary of Key Concepts and Patterns Covered

Our coverage of pattern matching in C# has covered both the fundamentals and advanced concepts. Here are a few highlights:

Introduction to Pattern Matching: The basics, syntax, and benefits of pattern matching in C#.

Constant Pattern Matching: The concept of constant pattern matching simplifies conditional logic by matching specific constant values in patterns.

Type Pattern Matching: Matching based on type and subtype, including inheritance and interface implementations.

Property Pattern Matching: Structured data matching based on properties of objects, tuples, and named fields.

Switch Expressions: Pattern matching enhances traditional switch statements for concise and expressive code.

Recursive Pattern Matching: Matching nested structures and collections, optimizing performance, and handling complex data scenarios using recursive pattern matching.

Pattern Combinators: For complex matching conditions, pattern combinators combine patterns using logical operators (and, or not).

Pattern Matching in LINQ Queries: Data retrieval and manipulation using pattern matching in LINQ to Objects and LINQ to SQL.

Pattern Matching in Async Methods: Handling asynchronous results and exceptions with pattern matching in async methods.

Advanced Topics: Topics in the advanced section of this course include pattern matching with tuples, deconstruction when clauses in switch statements, nullable types, and option types.

Practical Tips for Mastering Pattern Matching in C#

Practice Regularly: Maintain proficiency by practicing pattern matching regularly in your code.

Explore Advanced Patterns: Solve real-world problems using complex patterns and combinators.

Review and Refactor: Improve readability and maintainability of existing code by reviewing and refactoring existing code.

Stay Updated: Update yourself on the latest C# language updates and community practices for pattern matching.

Closing Off

As a powerful feature, pattern matching in C# improves readability, flexibility, and maintainability. You'll be able to leverage pattern matching effectively in your C# projects if you master the concepts covered in this book—from basic patterns to advanced techniques. To further refine your skills and tackle complex programming challenges confidently, you must practice continuously, explore advanced patterns, and stay up to date with resources.

All the Code Examples used in this book can be found here on my GitHub Repository: https://github.com/ziggyrafiq/CSharp-Pattern-Matching-Examples also please do not forget to follow me on LinkedIn https://www.linkedin.com/in/ziggyrafiq/

C#Corner

OUR MISSION

Free Education is Our Basic Need! Our mission is to empower millions of developers worldwide by providing the latest unbiased news, advice, and tools for learning, sharing, and career growth. We're passionate about nurturing the next young generation and help them not only to become great programmers, but also exceptional human beings.

ABOUT US

CSharp Inc, headquartered in Philadelphia, PA, is an online global community of software developers. C# Corner served 29.4 million visitors in year 2022. We publish the latest news and articles on cutting-edge software development topics. Developers share their knowledge and connect via content, forums, and chapters. Thousands of members benefit from our monthly events, webinars, and conferences. All conferences are managed under Global Tech Conferences, a CSharp Inc sister company. We also provide tools for career growth such as career advice, resume writing, training, certifications, books and white-papers, and videos. We also connect developers with their potential employers via our Job board. Visit C# Corner

MORE BOOKS