Pattern matching in C# 7.0 introduces a more expressive and readable way to work with types and data. It allows you to match expressions against patterns and deconstruct types in a way that makes your code more concise and clear. Let's explore a few examples of pattern matching in C# 7.0.

**Example 1: Type Pattern**

The type pattern allows you to check the type of an expression and, if it matches, assign it to a variable. Here's an example:

using System;

class Program

{

static void Main()

{

object obj = "Hello, world!";

if (obj is string s)

{

Console.WriteLine($"String length: {s.Length}");

}

else

{

Console.WriteLine("The object is not a string.");

}

}

}

In this example, the obj variable is checked to see if it is of type string. If it is, it is assigned to the variable s, and the length of the string is printed. If it is not a string, a different message is printed.

**Example 2: Constant Pattern**

The constant pattern allows you to compare an expression against a constant value. Here's an example:

csharp

using System;

class Program

{

static void Main()

{

int number = 42;

switch (number)

{

case 1:

Console.WriteLine("Number is 1.");

break;

case 42:

Console.WriteLine("Number is 42.");

break;

default:

Console.WriteLine("Number is something else.");

break;

}

}

}

In this example, the number variable is checked against constant values in a switch statement. If it matches one of the constants, the corresponding message is printed.

**Example 3: Var Pattern**

The var pattern allows you to assign an expression to a variable without specifying its type explicitly. Here's an example:

csharp

using System;

class Program

{

static void Main()

{

object obj = 123;

switch (obj)

{

case int i:

Console.WriteLine($"Integer value: {i}");

break;

case string s:

Console.WriteLine($"String value: {s}");

break;

default:

Console.WriteLine("Unknown type.");

break;

}

}

}

In this example, the obj variable is checked against different types in a switch statement. If it matches one of the types, the corresponding message is printed. The var pattern allows for type inference and makes the code more concise.

**Example 4: Deconstruction Pattern**

The deconstruction pattern allows you to deconstruct tuples and custom types into their constituent parts. Here's an example with a tuple:

csharp

using System;

class Program

{

static void Main()

{

var point = (X: 3, Y: 4);

if (point is (int x, int y))

{

Console.WriteLine($"Point coordinates: X = {x}, Y = {y}");

}

}

}

In this example, the point tuple is deconstructed into its constituent parts x and y, and their values are printed.

**Example 5: Combining Patterns**

You can combine different patterns to create more complex pattern matching. Here's an example:

csharp

using System;

class Program

{

static void Main()

{

object obj = (X: 3, Y: 4);

if (obj is (int x, int y))

{

Console.WriteLine($"Tuple coordinates: X = {x}, Y = {y}");

}

else if (obj is int i)

{

Console.WriteLine($"Integer value: {i}");

}

else

{

Console.WriteLine("Unknown type.");

}

}

}

In this example, the obj variable is checked against different patterns, including a deconstructed tuple and an integer. The corresponding message is printed based on the pattern match.

Pattern matching in C# 7.0 enhances the expressiveness and readability of your code, allowing you to work with different types and data structures more effectively.

Pattern matching in C# 7.0 brings several benefits that enhance the readability, maintainability, and expressiveness of your code. Here are some of the key advantages:

**1. Simplified Type Checking and Casting**

Pattern matching allows you to check the type of an object and cast it in a single, concise statement. This reduces boilerplate code and makes your intentions clearer.

**Example:**

csharp

// Without pattern matching

if (obj is string)

{

string s = (string)obj;

Console.WriteLine(s.Length);

}

// With pattern matching

if (obj is string s)

{

Console.WriteLine(s.Length);

}

**2. Enhanced Readability**

By using pattern matching, your code can become more readable and intuitive. It clearly expresses the patterns you expect and the actions you want to take based on those patterns.

**3. Reduced Boilerplate Code**

Pattern matching reduces the need for repetitive and verbose code, making your codebase cleaner and easier to maintain.

**Example:**

csharp

// Without pattern matching

if (obj is int)

{

int i = (int)obj;

// Use i

}

else if (obj is string)

{

string s = (string)obj;

// Use s

}

// With pattern matching

switch (obj)

{

case int i:

// Use i

break;

case string s:

// Use s

break;

}

**4. More Expressive Switch Statements**

Pattern matching enhances switch statements, allowing them to handle more complex scenarios with ease. You can match on types, constants, and even use additional conditions.

**Example:**

csharp

switch (obj)

{

case int i when i > 0:

Console.WriteLine("Positive integer");

break;

case int i when i <= 0:

Console.WriteLine("Non-positive integer");

break;

case string s:

Console.WriteLine($"String value: {s}");

break;

default:

Console.WriteLine("Unknown type");

break;

}

**5. Deconstruction Support**

Pattern matching supports deconstruction, allowing you to break down complex objects into their constituent parts seamlessly. This is particularly useful for working with tuples and custom types.

**Example:**

csharp

if (obj is (int x, int y))

{

Console.WriteLine($"Coordinates: X = {x}, Y = {y}");

}

**6. Improved Error Handling**

Pattern matching can be used to handle exceptions more effectively by matching specific conditions and types of exceptions.

**7. Maintainability**

With pattern matching, your code is easier to extend and modify. Adding new patterns or conditions can be done with minimal changes to existing code.

Overall, pattern matching in C# 7.0 makes your code more expressive, concise, and easier to read and maintain. It provides a powerful way to handle various scenarios and data structures, enhancing the overall quality of your code.