<Coding𝄞onata />

16 Types of Pattern Matching in C#

Aram Tchekrekjian

@AramT87

Pattern Matching

Pattern matching is a functional programming feature that already exists in other popular languages such as F#, Scala, Rust, Python, Haskell, Prolog and many other languages

Pattern Matching

It is used to provide a way to test expressions for some conditions while testing the types

Pattern Matching was introduced in C# 7

@AramT87

Pattern Matching - Benefits

- Type-Testing
- Nullable-type checking
- Type casting and assignment
- High readability
- Concise syntax
- Less convoluted code

Pattern Matching - Usages

'is' expressions

'switch' expressions

Pattern Matching Types

C# 7

- Type Pattern
- Declaration Pattern
- Constant Pattern
- Null Pattern
- Var Pattern

Pattern Matching Types

C# 8

- Property Pattern
- Discard Pattern
- Positional Pattern
- Tuple Pattern

Pattern Matching Types

C# 9

- Enhanced Type Pattern
- Relative Pattern
- Logical Pattern (Combinators)
- Negated Null Constant Pattern
- Parenthesized Pattern

Pattern Matching Types

C# 10

- Extended Property Pattern

C# 11

- List Pattern

Type Pattern

Type-testing for the expression

```
public bool IsProductFood(object product)
{
    return product is FoodModel;
}
```

Declaration Pattern

Type-testing as well as assignment to variable after a successful match of the expression

```
public bool IsProductFoodThatRequiresRefrigeration(object product)
{
    return product is FoodModel food &&
           RequiresRefrigeration(food.StorageTemperature);

}
```

Constant Pattern

Testing versus a constant value which can include int, float, char, string, bool, enum, field declared with const, null

```
public bool IsFreshProduce(FoodModel food)
{
    return food?.Category?.ID is
      (int) ProductCategoryEnum.FreshProduce;
}
```

@AramT87

Null Pattern

Check if a reference or nullable type is null

```
public bool FoodDoesNotExist(FoodModel food)
{
    return food is null;
}
```

Var Pattern

Similar to the type pattern, the var pattern matches an expression and checks for null, as well as assigns a value to the variable.
The var type is declared based on the matched expression’s compile-time type.

```
public bool IsProductFoodThatRequiresRefrigeration(FoodModel food)
{
    return GetFoodStorageRequirements(food) is var storageRequirement &&
           storageRequirement is StorageRequirementEnum.Freezer;
}
```

Property Pattern

Pattern matching using object members rather than variables to match the given conditions.

```
public bool IsOrganicFood(FoodModel food)
{
    return food is
    {
        NonGMO: true,
        NoChemicalFertilizers: true,
        NoSyntheticPesticides: true
    };
}
```

@AramT87

Discard Pattern

Using the discard operator _ to match anything including null.
In the below example, if you pass a food object with storageTemperature as 40, you get an exception

```
public int GetFoodStorageTemperature(StorageRequirementEnum storageRequirement) =>
  storageRequirement switch
{
    StorageRequirementEnum.Freezer => -18,
    StorageRequirementEnum.Refrigerator => 4,
    StorageRequirementEnum.RoomTemperature => 25,
    _ => throw new InvalidStorageRequirementException()
};
```

@AramT87

Positional Pattern

Mainly used with struct types, leverages a type's deconstrutor to match a pattern according to the values position in the deconstructor

```
public bool IsFreeFood(FoodModel food)
{
    return food.Price is (0, _);
}
```

Tuple Pattern

A special derivation from the positional pattern where you can test multiple properties of a type in the same expression, using tuples

```
public string GetFoodDescription(FoodModel food) =>
  (food.NonGMO, food.Category.ID) switch
{
    (true, (int)ProductCategoryEnum.FreshProduce) => "Non-GMO Fresh Product",
    (true, (int)ProductCategoryEnum.Dairy) => "Non-GMO Dairy",
    (false, (int)ProductCategoryEnum.Meats) => "GMO Meats. Avoid!",
    (_, _) => "Invalid Food Group"
};
```

‘Enhanced’ Type Pattern

You can do type checking in switch expressions without using the discards with each type

```
public string CheckValueType(object value) => value switch
{
    int => "This is an integer number",
    decimal => "This is a decimal number",
    double => "This is a double number",
    _ => throw new InvalidNumberException(value)
};
```

Relational Pattern

A relational pattern allows applying the relational operators > < >= <= to match patterns versus constants or enum values

```
public StorageRequirementEnum GetFoodStorageRequirements(FoodModel food) =>
  food.StorageTemperature switch
{
    <= -18 => StorageRequirementEnum.Freezer,
    >= 2 and < 6 => StorageRequirementEnum.Refrigerator,
    > 6 and < 30 => StorageRequirementEnum.RoomTemperature,
    _ => throw new InvalidStorageRequirementException(food.StorageTemperature)
};
```

Logical Pattern

This represents the set of negation, conjunctive, disjunctive; not, and, or respectively, used to combine patterns and apply logical conditions on them.
Together these are called the pattern combinators

```
public bool RequiresRefrigeration(ProductCategoryEnum productType)
{
    return productType is ProductCategoryEnum.Dairy or ProductCategoryEnum.Meats;
}
```

Negated Null Constant Pattern

Checks expression for not null value

```
public bool BlogExists(BlogModel blog)
{
    return blog is not null;
}
```

Parenthesized Pattern

This allows the use of parenthesis to control the order of execution and to group logical expressions together. Mainly used with pattern combinators

```
public bool RequiresRefrigeration(int storageTemperature)
{
    return storageTemperature is > 1 and (< 6);
}
```

Extended Property Pattern

In C# 10, the nested properties matching syntax issue was addressed with the introduction of the Extended Property Pattern, syntax to use nested properties in pattern matching is now clean and concise.

```
public bool RequiresRefrigeration(FoodModel food)
{
    return food is
    {
        Category.ID: (int)ProductCategoryEnum.Dairy or
                     (int)ProductCategoryEnum.Meats
    };
}
```

List Pattern

The list matching pattern is the latest addition to the great set of pattern matching in C# 11.
With list pattern you can match a list or an array with a set of sequential elements

```
public (int?, int?) FindNumberOneAndNumberFour()
{
    int[] numbers = { 1, 2, 3, 4, 5 };
    // Match if 2nd value is anything, 3rd is greater than or equal 3, fifth is 5
    if (numbers is [var numberOne, _, >= 3, int numberFour, 5])
    {
        return (numberOne, numberFour);
    }
    return (null, null);
}
```

@AramT87

Pattern Matching in C#

C# 7

Declaration Pattern
Type Pattern
Constant Pattern
Null Pattern
Var Pattern

C# 8

Property Pattern
Discard Pattern
Positional Pattern
Tuple Pattern

C# 9

‘Enhanced’ Type Pattern
Relative Pattern
Logical Pattern
Negated Null Constant Pattern
Parenthesized Pattern

C# 10

Extended Property Pattern

C# 11

List Pattern

Found this useful?

Consider Reposting

Thank You

Follow me for more content

Aram Tchekrekjian

Microsoft® MVP Most Valuable Professional

AramT87

CodingSonata.com/newsletters

<Coding Sonata />